USENIX

conference

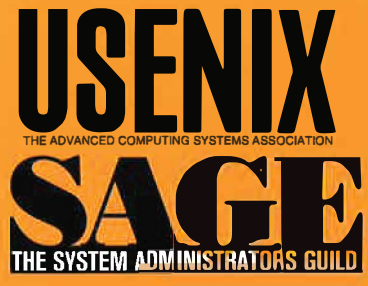
proceedings

LISA XVII

Seventeenth Large Installation Systems Administration Conference

San Diego, California, USA
October 26–31, 2003

Sponsored by **The USENIX Association** and **SAGE, the System Administrators Guild**



LISA XVII: 17th Large Installation Systems Administration Conference

San Diego, CA, Oct. 26–31, 2003

For additional copies of these proceedings contact

USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA 94710 USA
Telephone: +1 510-528-8649
http://www.usenix.org
<office@usenix.org>

Past LISA Conferences

Large Installation Systems Admin. I Workshop	1987	Phildelphia, PA
Large Installation Systems Admin. II Workshop	1988	Monterey, CA
Large Installation Systems Admin. III Workshop	1989	Austin, TX
Large Installation Systems Admin. IV Conference	1990	Colorado Spgs, CO
Large Installation Systems Admin. V Conference	1991	San Diego, CA
Systems Administration VI Conference	1992	Long Beach, CA
Systems Administration VII Conference	1993	Monterey, CA
Systems Administration VIII Conference	1994	San Diego, CA
Systems Administration IX Conference	1995	Monterey, CA
Systems Administration X Conference	1996	Chicago, IL
Systems Administration XI Conference	1997	San Diego, CA
Systems Administration XII Conference	1998	Boston, MA
Systems Administration XIII Conference	1999	Seattle, WA
Systems Administration XIV Conference	2000	New Orleans, LA
Systems Administration XV Conference	2001	San Diego, CA
Systems Administration XVI Conference	2002	Philadelphia, PA

ISBN 1-931971-15-3

USENIX Association

Proceedings of the Seventeenth Large Installation Systems Administration Conference (LISA XVII)

October 26-31, 2003
San Diego, CA, USA

ACKNOWLEDGMENTS

PROGRAM CHAIR
Æleen Frisch, *Exponential Consulting*

PROGRAM COMMITTEE
David N. Blank-Edelman, *Northeastern Univ.*
Gerald Carter, *HP/Samba Team*
Alva Couch, *Tufts University*
Michael Gilfix, *IBM*
Joshua Jensen, *IBM*
Douglas P. Kingston, *Deutsche Bank, London*
Mario Obejas, *Raytheon*
Joshua S. Simon, *Consultant*
David Williamson, *Certainty Solutions*
Elizabeth D. Zwicky, *Great Circle Associates*

SCRIBE
David Hoffman, *Stanford University*

COPY EDITORS
Jeff Allen, *Tellme Networks*
Æleen Frisch, *Exponential Consulting*

INVITED TALKS COORDINATORS
Esther Filderman, *Pitt. Supercomputing Ctr.*
Deeann Mikula, *Consultant*

NETWORKING IT COORDINATOR
William LeFebvre, *CNN Internet Technologies*

SECURITY IT COORDINATOR
Lynda True, *Northrop-Grumman*

GURU-IS-IN COORDINATOR
Lee Damon, *University of Washington*

WORK-IN-PROGRESS COORDINATOR
David Hoffman, *Stanford University*

TRAINING PROGRAM COORDINATOR
Daniel V. Klein, *USENIX Association*

WORKSHOP COORDINATOR
John Orthoefer, *Tufts University*

CONFIGURATION WORKSHOP
Paul Anderson, *University of Edinburgh*

SYSADMIN EDUCATION WORKSHOP
John Sechrest, *PEAK, Inc.*
Curt Freeland, *Notre Dame University*

ENTERPRISE INFRASTRUCTURE WORKSHOP
Steve Traugott, *Infrastructures.Org*

AFS WORKSHOP
Esther Filderman, *Pitt. Supercomputing Ctr.*
Derrick Brashear, *Carnegie Mellon University*
Gary Zaccheis, *MIT*

ADVANCED TOPICS WORKSHOP
Adam Moskowitz, *Menlo Computing*
Rob Kolstad, *SAGE*

CONFERENCE ADVISOR & PROCEEDINGS TYPESETTING
Rob Kolstad, *SAGE*

USENIX ASSOCIATION
Ellie Young, *Executive Director*
Paula Larink, *Conference Director*
Cat Allman, *Sales and Marketing Director*
Jane-Ellen Long, *IS/Production Director*
Jennifer Desimone, *Conference Manager*
Marci Chase, *Conference Manager*
Tony Del Porto, *System Administrator*

REFEREED PAPER EXTERNAL REVIEWERS

Jeff Allen, *Tellme Networks*
Paul Anderson, *Edinburgh University*
Steve Armijo, *University of New Mexico*
Kenytt Avery, *Willing Minds*
Rocky Bernstein
Melissa Binde, *Amazon.com*
Ian Michael Bland, *University of Reading*
Derrick Brashear, *Carnegie Mellon University*
Mark Burgess, *University College Oslo*
Gerald Carter, *HP/Samba Team*
Alva Couch, *Tufts University*
Crispin Cowan, *Immunix Secured Solutions*
Paul Delug, *American Physical Society*
Sven Dietrich, *CERT*
David Blank-Edelman, *Northeastern University*
Scott Fendley, *University of Arkansas*
Michael Gilfix, *IBM*
Ray Hiltbrand, *E*Trade Financial*
David Hoffman, *Stanford University*
David Homoney, *Federal Transit Adminstration*
Doug Hughes, *Auburn University*
Joshua Jensen, *IBM*
Jeremy Jethro, *@stake*
Alexander Keller, *IBM*
Doug Kingston, *Deutsche Bank*
Tom Limoncelli, *Lumeta*
John Rowan Littell, *Earlham College*
Adrian Filipi-Martin, *Ubergeeks Consulting*
Ellen Mitchell, *Texas A&M University*
Tejas Naik, *UC Irvine*
Ron Nelson, *Ron Nelson Professional Services*
Mario Obejas, *Raytheon*
John Orthoefer, *Tufts University*
Will Partain, *Verilab*
Tom Perrine, *Sony Computer Electronics of America*
Dustin Puryear, *Puryear Information Technology*
S. Raj Rajagopalan, *Telcordia Technologies*
David Ressman, *University of Chicago*
Kam Salisbury, *Digital Wave Technologies*
Froda-Eika Sandnes, *University College Oslo*
Phil Scarr, *GE Industrial Systems*
Juergen Schoenwaelder, *International University Bremen*
Vlakkies Schreuder, *University of Colorado*
John Sechrest, *PEAK Internet*
John Sellens, *General Concepts*
David Harnick-Shapiro, *UC Irvine*
Joshua Simon, *Consultant*
Sigmund Straumsmes, *University College Oslo*
Jenn Strum, *Hamilton College*
Lynda True, *Northrop-Grumman*
Steven Tylock, *Lynx Technologies*
Todd Watson, *Southwestern University*
Jeffrey Whitehead, *Epinions.com*
David Williamson, *Certainty Solutions*
Garry Zacheiss, *MIT*
Elizabeth Zwicky, *Great Circle Associates*

CONTENTS

Acknowledgments ii

Author Index vi

Message from the Program Chair vii

Opening Remarks

Wednesday (8:45-9:00 am) **Chair: Æleen Frisch**

Keynote

Wednesday (9:00-10:30 am) **Speaker: Paul Kilmartin**

Administering Essential Services

Wednesday 11:00 am-12:30 pm **Gerald Carter**

Radmind: The Integration of Filesystem Integrity Checking with Filesystem Management 1
Wesley D. Craig and Patrick M. McNeal, The University of Michigan

Further Torture: More Testing of Backup and Archive Programs 7
Elizabeth D. Zwicky

An Analysis of Database-Driven Mail Servers 15
Nick Elprin and Bryan Parno, Harvard College

Information and Content Management

Wednesday 2:00-3:30 pm **Alva Couch**

A Secure and Transparent Firewall Web Proxy 23
Roger Crandell, James Clifford, and Alexander Kent, Los Alamos National Laboratory

Designing, Developing, and Implementing a Document Repository 31
Joshua Simon, Consultant; Liza Weissler, METI

DryDock: A Document Firewall 41
Deepak Giridharagopal, The University of Texas at Austin

System and Network Monitoring

Wednesday 4:00-5:30 pm **Michael Gilfix**

Run-time Detection of Heap-based Overflows 51
William Robertson, Christopher Kruegel, Darren Mutz, and Fredrik Valeur, University of California, Santa Barbara

Designing a Configuration Monitoring and Reporting Environment 61
Xev Gittler and Ken Beer, Deutsche Bank

New NFS Tracing Tools and Techniques for System Analysis 73
Daniel Ellard and Margo Seltzer, Harvard University

Difficult Tasks Made Easier

Thursday 9:00-10:30 am — Elizabeth Zwicky

EasyVPN: IPsec Remote Access Made Easy 87
Mark C. Benvenuto and Angelos D. Keromytis, Columbia University

The Yearly Review, or How to Evaluate Your Sys Admin 95
Carrie Gates and Jason Rouse, Dalhousie University

Peer Certification: Techniques and Tools for Reducing System Admin Support Burdens while Improving Customer Service 105
Stacy Purcell, Sally Hambridge, David Armstrong, Tod Oace, Matt Baker, and Jeff Sedayao, Intel Corp.

Emerging Theories of System Administration

Thursday 11:00 am-12:30 pm — Æleen Frisch

ISconf: Theory, Practice, and Beyond 115
Luke Kanies, Reductive Consulting, LLC

Seeking Closure in an Open World: A Behavioral Agent Approach to Configuration Management 125
Alva Couch, John Hart, Elizabeth G. Idhaw, and Dominic Kallas, Tufts University

Archipelago: A Network Security Analysis Tool 149
Tuva Stang, Fahimeh Pourbayat, Mark Burgess, Geoffrey Canright, Kenth Engø, and Åsmund Weltzien, Oslo University College

Configuration Management: Tools and Techniques

Friday 9:00-10:30 am — Michael Gilfix

STRIDER: A Black-box, State-based Approach to Change and Configuration Management and Support 159
Yi-Min Wang, Chad Verbowski, John Dunagan, Yu Chen, Helen J. Wang, Chun Yuan, and Zheng Zhang, Microsoft Research

CDSS: Secure Distribution of Software Installation Media Images in a Heterogeneous Environment 173
Ted Cabeen, Impulse Internet Services; Job Bogan, Consultant

Virtual Appliances for Deploying and Maintaining Software 181
Constantine Sapuntzakis, David Brumley, Ramesh Chandra, Nickolai Zeldovich, Jim Chow, Monica S. Lam, and Mendel Rosenblum, Stanford University

Configuration Management: Analysis and Theory

Friday 11:00 am-12:30 pm — Michael Gilfix

Generating Configuration Files: The Directors Cut 195
Jon Finke, Rensselaer Polytechnic Institute

Preventing Wheel Reinvention: The psgconf System Configuration Framework 205
Mark D. Roth, University of Illinois at Urbana-Champaign

SmartFrog meets LCFG: Autonomous Reconfiguration with Central Policy Control 213
Paul Anderson, University of Edinburgh; Patrick Goldsack, HP Research Laboratories; Jim Paterson, University of Edinburgh

Network Administration

Friday 2:00-3:30 pm — **David Williamson**

Distributed Tarpitting: Impeding Spam Across Multiple Servers 223
Tim Hunter, Paul Terry, and Alan Judge, eircom.net

Using Service Grammar to Diagnose BGP Configuration Errors 237
Xiaohu Qie, Princeton University; Sanjai Narain, Telcordia Technologies

Splat: A Network Switch/Port Configuration Management Tool 247
Cary Abrahamson, Michael Blodgett, Adam Kunen, Nathan Mueller, and David Parter, University of Wisconsin – Madison

AUTHOR INDEX

Cary Abrahamson 247
Paul Anderson 213
David Armstrong 105
Matt Baker 105
Ken Beer 61
Mark C. Benvenuto 87
Michael Blodgett 247
Job Bogan 173
David Brumley 181
Mark Burgess 149
Ted Cabeen 173
Geoffrey Canright 149
Ramesh Chandra 181
Yu Chen 159
Jim Chow 181
James Clifford 23
Alva Couch 125
Wesley D. Craig 1
Roger Crandell 23
John Dunagan 159
Daniel Ellard 73
Nick Elprin 15
Kenth Engø 149
Jon Finke 195
Carrie Gates 95
Deepak Giridharagopal 41
Xev Gittler 61
Patrick Goldsack 213
Sally Hambridge 105
John Hart 125
Tim Hunter 223
Elizabeth G. Idhaw 125
Alan Judge 223
Dominic Kallas 125
Luke Kanies 115
Alexander Kent 23
Angelos D. Keromytis 87
Christopher Kruegel 51
Adam Kunen 247
Monica S. Lam 181
Patrick M. McNeal 1
Nathan Mueller 247
Darren Mutz 51
Sanjai Narain 237
Tod Oace 105
Bryan Parno 15
David Parter 247
Jim Paterson 213
Fahimeh Pourbayat 149
Stacy Purcell 105
Xiaohu Qie 237
William Robertson 51
Mendel Rosenblum 181
Mark D. Roth 205
Jason Rouse 95
Constantine Sapuntzakis 181
Jeff Sedayao 105
Margo Seltzer 73
Joshua Simon 31
Tuva Stang 149
Paul Terry 223
Fredrik Valeur 51
Chad Verbowski 159
Helen J. Wang 159
Yi-Min Wang 159
Liza Weissler 31
Åsmund Weltzien 149
Chun Yuan 159
Nickolai Zeldovich 181
Zheng Zhang 159
Elizabeth D. Zwicky 7

Message from the Program Chair

These Proceedings from the 2003 LISA Conference contain the complete texts of the refereed papers presented at that conference. They appear in this volume in the order in which they were presented.

This year's papers are an exceptionally fine set. The 24 of them were culled from a set of 90 submissions. Together they make significant contributions both to system administration practice – presenting approaches and techniques relevant to what system administrators do in their jobs – and to the emerging system and network administration academic subdiscipline, including several related to the theory of system administration.

Readers will probably want to begin with the papers most closely related to their own needs and interests. To make this easier, the following bullets identify several classes of papers:

- Tools to Aid/Automate System Administration: Craig, Giridharagopal, Gittler, Ellard, Kaines, Stang, Wang, Cabeen, Finke, Roth, Abrahamson
- Comparative Analysis of Current Tools/Techniques: Zwicky, Elprin, Ellard, Anderson, Robertson
- Theory of System Administration: Couch, Finke, Roth, Anderson, Stang
- Infrastructure/Configuration Management Theory and Techniques: Kaines, Wang, Cabeen, Finke, Roth, Anderson, Sapuntzakis
- Networking: Benvenuto, Ellard, Hunter, Qie, Abrahamson
- Managing People and Information: Gates, Sedayao, Simon, Crandell
- Email and Web Services: Elprin, Giridharagopal, Hunter

We hope that you find these works useful and enjoyable, and that you consider presenting your own work at future LISA conferences.

Æleen Frisch
Program Chair

Radmind: The Integration of Filesystem Integrity Checking with Filesystem Management

Wesley D. Craig and Patrick M. McNeal – The University of Michigan

ABSTRACT

We review two surveys of large-scale system management, observing common problems across tools and over many years. We also note that filesystem management tools interfere with filesystem integrity checking tools. Radmind, an integrated filesystem management and integrity checking tool, solves many of these problems. Radmind also provides a useful platform upon which to build further large-scale, automatic system management tools.

Introduction

We present "Radmind," an open source suite of Unix command-line tools and a server designed to remotely administer the filesystems of multiple, diverse Unix machines. At its core, Radmind checks filesystem integrity. Like Tripwire [Spafford], it is able to detect changes to any managed filesystem object. Radmind, though, goes further than just integrity checking. Files are stored on a remote server so that Radmind can reverse changes if filesystem integrity is lost. Changes can be made to managed machines by updating files on the Radmind server.

Each managed machine may have its own *load-set* composed of multiple, layered *overloads*. Overloads consist of a list of filesystem objects and any associated file data. This allows, for example, the operating system to be described separately from software packages.

One of the great strengths of Radmind is that it does not enforce a management ideology. It allows a system administrator to manage many machines using roughly the same skill set she already uses to manage one machine. It also provides a useful filesystem integrity check that neither undermines nor is undermined by filesystem management.

State of the Art

In the recently published "Experiences and Challenges of Large-Scale System Configuration," Anderson, et al. [Anderson 03] review several large-scale system management tools: SUE [CERN], cfengine [Burgess], LCFG [Anderson 02], and several ad hoc solutions. They identify several common problems:

- There is no method to determine that a running node has diverged from its specification. Without this information, the only option is to reinstall.
- The complexity of existing configuration tools requires extensive knowledge, above what may be expected of a newly hired but experienced system administrator.
- New operating systems and new operating system versions are a major problem.
- Server management is typically ad hoc, not part of the large-scale configuration management system.
- There is no support for disconnected nodes, e.g., laptops.

Seven years earlier in "An Analysis of UNIX System Configuration," [Evard 97] Evard describes many of these same problems. He also notes that large-scale configuration management has been "an area of exploration for at least ten years." While many sophisticated tools have been made available in the last seventeen years, the same problems are still prevalent.

Also of note is the degree to which filesystem integrity checking conflicts with these system management tools. Groups like SANS and CERT list filesystem integrity checking as one of the basic procedures that all system administrators should use to help secure their computers. However, if a cluster is running both a filesystem integrity tool for intrusion detection and, e.g., rsync [Tridgell] for software updates, each time an update occurs, every machine will report a security event. For large clusters, these reports are noise in which real problems may be lost [Arnold]. In order to update the managed systems without triggering security events, the system management tool must be aware of (or integrated with) the intrusion detection tool.

Integrating intrusion detection with system management affords several additional capabilities that make it attractive. While filesystem integrity checking tools detect the changes that an attacker has made to the filesystem, they can also detect certain types of data corruption caused by operating system and hardware errors, e.g., a flaky disk driver or controller. Because filesystem integrity checking tools scan virtually all of the nonvolatile filesystem, they detect the changes that the system administrator makes as well. For example, they can detect what "make install" has done, or RPM [Bailey] , or even "vi." Once captured,

this information can be propagated by the system administrator to manage an entire cluster of machines.

Synctree [Lockard], also written at the University of Michigan, is able to leverage filesystem integrity information for the purpose of system management. However, Synctree requires AFS as a file transfer mechanism and requires AFS system:administrator ("root" in AFS) access. In a non-AFS environment, it would be an excessive burden to bring up an AFS cell just for system management. Moreover, a non-AFS-dependent mechanism would be required to securely manage the AFS servers.

Despite Synctree's shortcomings and minimal adoption, it demonstrates that combining integrity checking with filesystem management is a viable methodology. Below, we demonstrate the advantages of this methodology and how it can be used to resolve the issues that have been plaguing automatic system administration for nearly two decades.

Divergence

Because Radmind includes a filesystem integrity checking tool, fsdiff (filesystem difference), that traverses the nonvolatile portions of the filesystem, it is possible at any time to compare a live node with a well-defined specification. Any modifications, additions or deletions to the local filesystem are reflected in the output of fsdiff. Therefore, it is possible to determine a priori the correctness of a node. When a node diverges from its specification the system administrator has the option to audit the *transcript* (Figure 1) of differences.

Without this information, the system is always running in an indeterminate state. When the system administrator responds to aberrant behavior from a particular node, she cannot distinguish intrusion, hardware failure, software bugs, staff error, etc., from system divergence. The only way to eliminate system divergence as a possible source of error is to reinstall.

Radmind, on the other hand, can utilize a difference transcript to return the node to its specification. A separate tool, lapply (loadset apply), reads a transcript and modifies the filesystem accordingly. lapply removes, modifies and creates filesystem objects, downloading files from a Radmind server as necessary. Since the transcript lists only differences, lapply makes the minimum changes required.

Interestingly, if the specification has changed, the same procedure can be used to install updates onto nodes. The specification consists of a series of transcripts and a listing of their precedence, known as a *command file* (Figure 2). The ktcheck (command file and transcript check) tool calculates a checksum of the local copy of the command file and transcripts, compares it with a checksum returned by the Radmind server, and optionally updates the local copies. When ktcheck indicates that the command file or transcripts have been updated, differences from the specification are treated as intentional updates rather than anomalies in the filesystem. The ability to differentiate a system update from a security event makes the security event reporting valuable, particularly in a large-scale environment.

Complexity

Meta-configuration tools like LCFG, PAN [Cons], and cfengine each implement their own declarative language to manage the contents of configuration files. They do not directly address the management of the remainder of the filesystem. For example, cfengine deployments use other tools, e.g., RPM, to manage system binaries. In order to use cfengine with

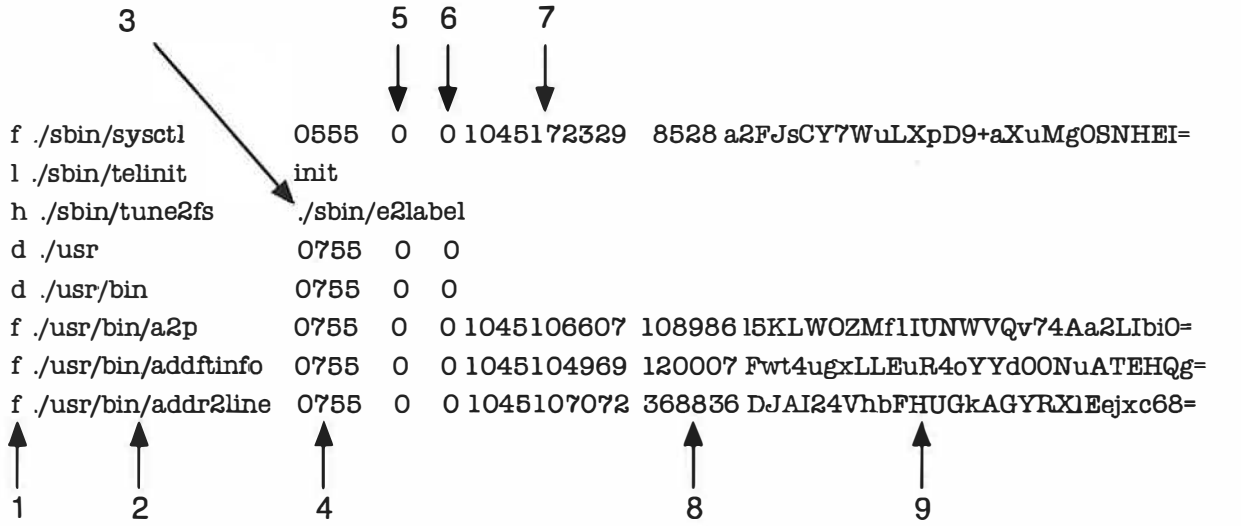


Figure 1: Fragment of a Linux From Scratch base transcript. 1. File type. Shown are regular files (f), a symbolic link (l), a hard link (h), and directories (d). 2. Pathname. Transcripts are sorted by pathname. 3. Link target. 4. Mode, listed in octal. 5. Owner ID. 6. Group ID. 7. mtime. 8. Size in bytes. 9. Base64 encoded SHA-1 cryptographic checksum.

RPM, all software must be packaged, even if it is built from source. Unfortunately, the system administrator is then required to learn two complex applications and is still unable to fully verify the integrity of the non-volatile filesystem.

The above meta-configuration tools have semantic knowledge of the contents of configuration files. This implicit knowledge forces the system administrator to use the tool's language to update or create a configuration file, preventing her from using her preferred method. If the system administrator uses a tool like Webmin [Cameron] in a cfengine environment, cfengine will simply overwrite her changes. There is no provision in cfengine for capturing complex changes, saving them, and automatically incorporating them into a given configuration.

In a Radmind environment, fsdiff is able to detect any changes, regardless of how the changes were made. This permits the system administrator to use whatever tool is appropriate to make changes. A separate tool, lcreate (loadset create), reads fsdiff's output and stores the transcript and any associated files to a Radmind server (Figure 3). Once the loadset is on the server, the system administrator can either merge the new loadset into an existing loadset or treat it as an overload. By not merging, the system administrator can easily test new loadsets while retaining the option to back them out. This allows a system administrator to manage a large number of diverse machines using normal system administration tools plus Radmind without requiring additional scripting or programming.

Portability

During an operating system upgrade, configuration files, system binaries, and the kernel are updated. Tools like cfengine are unable to directly manage system binaries and the kernel. This makes operating system upgrades challenging. Since cfengine's classes, scripts and edits contain the semantics of the system configuration files, adding support for new operating systems often involves writing substantial new code. Current meta-configuration tools support a limited number of operating systems, and a system administrator must wait to deploy a new operating system until its semantics have been incorporated. Once a version of cfengine supports the target operating system, the system administrator still must revise her site-specific cfengine scripts and configuration files.

By contrast, Radmind has no knowledge of the contents of files. Files are instead treated as simple byte-streams. Porting Radmind to new platforms is easy, typically requiring only recompilation. In unusual cases such as Mac OS X's HFS+ multi-forked files or Solaris "door" files, support for new filesystem objects may need to be added to the code base.

While treating managed files as byte-streams greatly improves portability, it also has some drawbacks. Some applications use a single file for configuration, e.g., inetd uses /etc/inetd.conf. With Radmind, two inetd.conf configurations that differ by only a single line require two separate files. In an environment with even moderate diversity, maintaining the common portions of many inetd.conf files is a chore. There are many solutions to this problem. xinetd [xinetd], for example, may load the contents of a directory for its configuration. This allows the common configuration to be shared amongst a large number of machines with the differences represented by separate whole files. Or, to use Radmind with traditional inetd, inetd's startup script could synthesize /etc/inetd.conf from a directory containing inetd.conf fragments.

A common rationale for encoding the semantics of configuration files into meta-configuration tools is the ability to make a single change and affect an entire infrastructure. For instance, one could specify that a

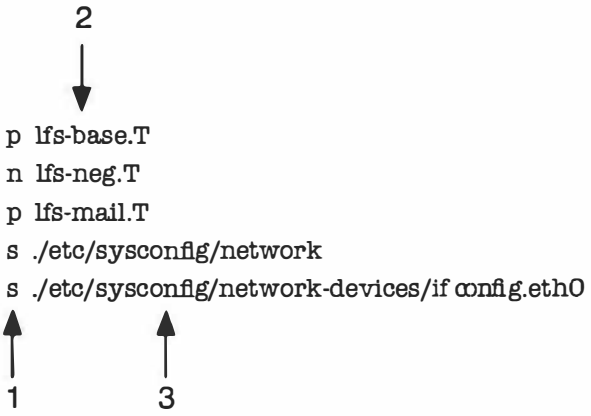


Figure 2: Command file for a Linux From Scratch mail server. 1. Transcript type. Listed are positive transcripts (p), a negative transcript (n), and special files (s). 2. Transcript name. 3. Special file pathname. Positive transcripts list the managed portion of the filesystem. Negative transcripts define the unmanaged portions of the filesystem. Transcripts are listed from lowest to highest precedence. The special transcript, built from the list of special files, always has the highest precedence.

machine should be an NTP server, and simultaneously that other machines running multiple operating systems should use that NTP server. This ultra-abstract configuration management is another source of complexity and non-portability inherent in meta-configuration tools. The problem described above is more easily managed by removing the configuration to a service location mechanism such as ZEROCONF [IETF], DHCP, or DNS. Evard makes a similar observation, advising system administrators to "migrate changes into the network and away from the host" [Evard 97].

Servers

Large-scale configuration management tools are most popular in computing clusters, labs, and other environments where hardware and software diversity is very low. However, even when widely deployed, these systems are often not used to manage servers. As an example, the 100 nodes of a computing cluster might be managed while the two or three supporting servers are built by hand. While it is easy to share one image for all of the cluster nodes, existing tools have difficulty managing the differences between servers and between cluster nodes and servers.

With Radmind, fsdiff is able to automatically detect and report the differences between machine specifications. A system administrator can install a server with the same base loadset as a compute node, install additional software required for, e.g., NTP, and capture those changes as an overload. This overload plus the cluster base loadset can then be applied to any machine to produce an NTP server.

When machines are exactly alike except for host-specific files, an overload is not necessary. Such machines can share one command file, listing the appropriate transcripts and any *special files*. When ktcheck runs, host-appropriate special files will be referenced. In this way, machines can share the bulk of their specifications while retaining their individuality.

Laptops

Laptops are particularly challenging due to their volatile network configuration. Mobile computers are on the network only intermittently. Moreover, when they are connected, the available bandwidth is often low. Configuration management tools are also challenged by the potentially weak security of the network and the fact that machines may need to be updated from any IP-address.

Since all Radmind functions are initiated by the managed machines, updates can be triggered by the user when he knows that the network will be available. Moreover, the tools fail gracefully when the network is unavailable. Radmind also minimizes network traffic in the case where the loadset has not changed. If changes are detected, Radmind transfers only the necessary files.

For network security, Radmind supports SSL-encrypted links. This allows nodes on insecure networks to be updated securely. SSL certificates can also be used

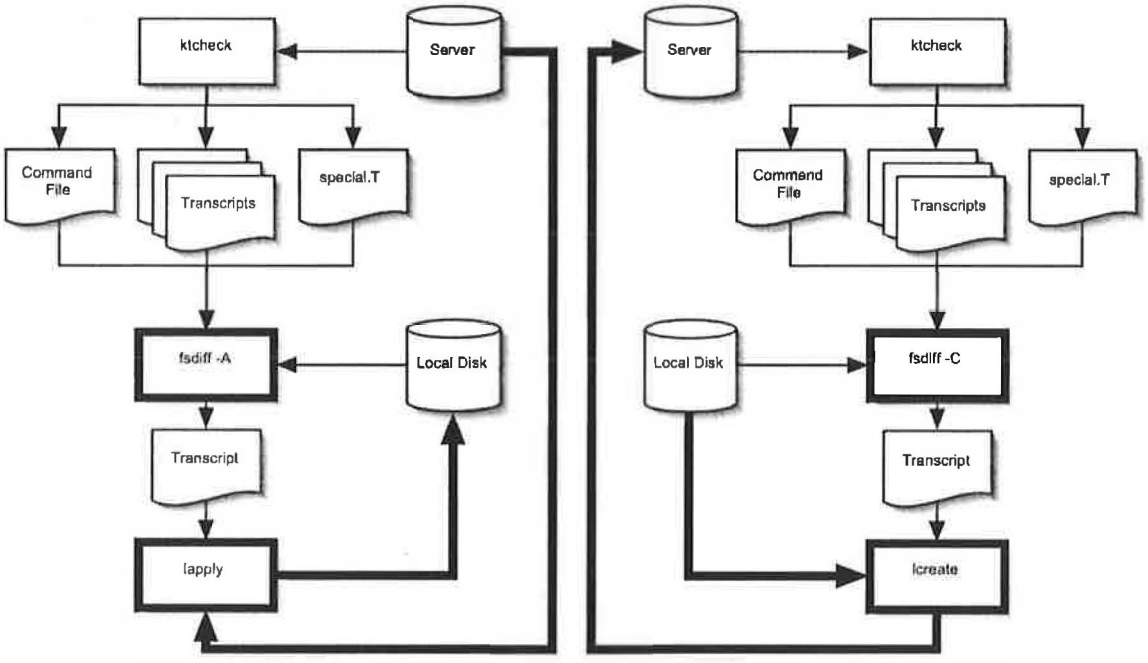


Figure 3: Overview of Radmind tools and data flow. Like many classic Unix tools, fsdiff's input and output are in the same format. fsdiff is able to output two types of transcripts, *apply-able* (left) and *create-able* (right). Apply-able transcripts describe the changes needed to make a filesystem match a specification. Create-able transcripts describe the changes needed to make a specification match a filesystem.

to authenticate both the Radmind server and the managed clients, regardless of DNS or IP-address variation.

Future Work

Radmind does not directly address the initial installation of machines. Instead, most system administrators install a minimal operating system on each machine before it can be managed with Radmind. More sophisticated solutions are possible, e.g., using PXE [Intel] or NetBoot [Apple]. In addition, work has been done to create a "Radmind CD." These CDs are built from a production base loadset and an auto-install overload. Using this methodology, creating new CDs to support additional hardware is trivial.

Since Radmind only manages the filesystem, other methodologies are required to manage the master boot record of a machine, processes, etc. Any one of these activities is relatively easy to layer on top of fsdiff, either before or after lapply is run. It is difficult to automatically determine all of the correct actions to take when large filesystem changes are made. Tools like sowhat [Couch] are helpful for determining interdependencies among filesystem objects, but again, leave much manual work to the system administrator. Automatically determining the correct order of process stops and starts is nontrivial. Like automatic system administration in general, this area is ripe for further exploration.

Conclusion

Radmind is an open source, complete solution to the large-scale filesystem management problem. Its external requirements are small, and the tools themselves are easy to use – there is no additional language to learn. Radmind's security is based on OpenSSL, both for SSL/TLS encryption and authentication and for support of cryptographic hash functions such as SHA-1 and RIPEMD-160. Radmind is freely available from http://radmind.org .

Author Information

Wesley D. Craig joined the University of Michigan in 1987, where he designed and wrote netatalk. He is currently the Senior IT Architect and Engineer for the University's Research Systems UNIX Group where he manages the team that runs the University's central LDAP directory, e-mail, and charged-for printing systems. Reach him electronically at wes@umich.edu .

Patrick M. McNeal earned his BSE in computer engineering from the University of Michigan. Upon graduation, he joined the University's Research Systems UNIX Group as a software engineer. He works on the Radmind project along with other e-mail-related services. Reach him electronically at mcneal@umich.edu .

References

[Anderson 02] Anderson, P. and A. Scobie, "LCFG – The Next Generation," *UKUUG Winter Conference*, http://www.lcfg.org/doc/ukuug2002.pdf , 2002.

[Anderson 03] Anderson, P., G. Beckett, K. Kavoussanakis, G. Mecheneau, and P. Toft, "Experiences and Challenges of Large-Scale System Configuration." *GridWeaver Project*, http://www.epcc.ed.ac.uk/gridweaver/WP2/report2.pdf , March, 2003,

[Apple] Apple Computer, Inc. http://www.apple.com .

[Arnold] E. Arnold, "The Trouble With Tripwire: Making a Valuable Security Tool More Efficient," http://www.securityfocus.com/infocus/1398 , June 6, 2001.

[Bailey] Bailey, E., "Maximum RPM (RPM)." MacMillan Publishing Company, August, 1997.

[Burgess] Burgess, M. and R. Ralston, "Distributed Resource Administration Using Cfengine," *Software: Practice and Experience*, Vol. 27, 1997.

[Camero] Camero, J., "Webmin: A Web-based System Administration Tool for Unix," Freenix 2000, *USENIX Annual Technical Conference*, 2000.

[CERN] CERN, *SUE*, http://wwwpdp.web.cern.ch/wwwpdp/ose/sue/doc/sue.html .

[Cons] Cons, L. and P. Poznanski, "Pan: A High-Level Configuration Language," *Proceedings LISA XVI*, USENIX Association, 2002.

[Couch] Couch, Alva L., "Global Impact Analysis of Dynamic Library Dependencies," *Proceedings LISA XV*, USENIX Association, 2001.

[Evard 97] Evard, R. "An Analysis of UNIX Machine Configuration." *Proceedings LISA XI*, USENIX Association, 1997.

[IETF] IETF, *Zeroconf Working Group*, http://www.ietf.org/html.charters/zeroconf-charter.html .

[Intel] Intel Corporation, "Preboot Execution Environment (PXE) Specification," September 20, 1999.

[Lockard] Lockard, J. and J. Larke, "Synctree for Single Point Installation, Upgrades, and OS Patches," *Proceedings LISA XII*, USENIX Association, 1998.

[RPM] RPM, http://www.rpm.org .

[Spafford] Kim, G. and E. Spafford, "Experiences with TripWire: Using Integrity Checkers for Intrusion Detection." *Proceedings System Administration, Networking, and Security, III*, Usenix Assoc., 1994.

[Tridgell] Tridgell, A. and P. Mackerras, "The rsync Algorithm," Australian National University, *TR-CS-96-05*, June, 1996.

[xinetd] *xinetd*, http://www.xinetd.org .

Further Torture: More Testing of Backup and Archive Programs

Elizabeth D. Zwicky

ABSTRACT

Every system administrator depends on some set of programs for making secondary copies of data, as backups for disaster recovery or as archives for long term storage. These programs, like all other programs, have bugs. The important thing for system administrators to know is where those bugs are.

Some years ago I did testing on backup and archive programs, and found that failures were extremely common and that the documented limitations of programs did not match their actual limitations. Curious to see how the situation had changed, I put together a new set of tests, and found that programs had improved significantly, but that undocumented failures were still common.

In the previous tests, almost every program crashed outright at some point, and dump significantly out-performed all other contenders. In addition, many programs other than the backup and archive programs failed catastrophically (most notably fsck on many platforms, and the kernel on one platform). Few programs other than dump backed up devices, making them incapable of backing up and restoring a functional operating system. In the new tests, crashes are rare, and more programs function as backups of entire systems, but failures in backup and archive programs are still widespread, and dump is not inherently more reliable than other programs.

Why Is Copying Files Hard?

You would think that the problem of writing backup and archive programs would have been solved some time ago. After all, the basic task has been performed for almost as long as computers have existed. But there are a number of problems.

Flexibility and Inflexibility

Filesystem design changes relatively fast, and human behavior changes even faster. Archive formats, on the other hand, mutate so slowly that many archive programs will attempt to not only read but also write tar archives in a format not significantly changed in the last 20 years. This is a format that allowed 100 characters for a filename (because it was invented in a world that had only recently given up 14 character filenames, and that had not yet reached auto-completing shells or graphical user interfaces).

The result is what amounts to a war between the filesystem people and the archive people. Just when you think you've invented an archive format that's satisfactorily flexible – it doesn't arbitrarily limit filenames, it can deal with any number of hard links, it can take any character you can dish out – somebody comes along and increases the size of inode numbers, device numbers, user names (which you may be storing as part of the ownership data), and then adds access control lists.

Combinatorics and the Contrariness of Users

My current set of test conditions involves tens of thousands of files, and it tests a fairly small number of conditions. It's quite possible that it misses a number of cases; for instance, what about bugs that are triggered by files with very long names and unusual permissions? Testing backup and archive programs is not an easy process, and it's not surprising that program authors have problems with it.

All this wouldn't matter much if people used a consistent set of filesystem features, but they don't. However improbable a combination may appear, somebody somewhere is going to use it for something. It may be pure accident; it may be the result of file sharing, where perfectly reasonable behavior on one system often becomes deeply weird behavior on another; it may be an automated system that ends up doing things a human wouldn't have thought of; or it may be human perversity. But one way or another, over time, the oddest corners of the filesystem will be explored.

Active Filesystems

Filesystems that are in use are not guaranteed to be in a consistent state. Worse yet, even if they are in a consistent state at the moment when you start to back them up, they will change during the process. Files will be created, deleted, or modified; whole directories may come and go. These problems are discussed in detail in Shumway [1].

The Tests

File Size

Historically, some archive programs have chosen to ignore empty files or directories. This is extremely annoying; empty files are often being used as indicator flags, and empty directories may be mount points. The test program therefore creates an empty file and an

empty directory to ensure that they can be backed up. It also creates a test file over four gigabytes to ensure that the program handles large files well.

Devices

In order to restore a system to a functional state, you need to be able to back up and restore devices. The testing program therefore creates a named pipe, a block device, and a character device. (I know of no reason why a program would back up block devices but not character devices, or vice versa; it does both purely for the sake of completeness.) It also creates a character device with major and minor device numbers 65025, if possible, in order to test handling of new longer device numbers.

Strange Names

Different filesystems impose different limitations on the character sets that can be used in filenames. Because "/" is used as a directory separator, it is illegal in filenames; because null is used as a string terminator, although it is theoretically legal it is not in general practical to insert it in a filename. Beyond that, some filesystems allow only 7-bit characters (standard ASCII, including annoyances like escape and bell, which makes your terminal beep), while others allow larger character sets that use 8-bit or larger characters.

There are circumstances in which files can end up having illegal filenames. In particular, some programs that share filesystems with other operating systems do not go through the local UNIX filesystem. Since "/" is not the directory separator in either traditional MacOS or Windows, while it is traditionally used to write dates with, these machines may write filenames with "/" in them. These problems are rare these days; programs which share filesystems now routinely translate directory separators so that a Macintosh file which contains a "/" is written on UNIX as ":", and a UNIX file with ":" in it is shared to a Macintosh with a "/", for instance. However, you should be aware that any program that writes to the disk device directly, which may include file sharing programs, is capable of writing files that the filesystem doesn't properly understand. Such files may be possible to back up (for instance, with "dump"), but they are unlikely to be possible to restore. These tests don't test truly illegal filenames.

Strange characters cause problems different ways depending on where they're found. For instance, some programs have problems with whitespace at the end or beginning of file names, but not in the middle; others handle directory names, symbolic link names, or symbolic link targets differently from file names. Therefore, the program runs through all 127 characters of 7-bit ASCII, creating the following files, directories, and links (<charnumber> is the decimal number corresponding to the character, and is included so that when things fail, you know which character caused the failure even if it is non-printable):

- Directories, each containing a file named "<charnumber>":
  - funnydirnames/a<char><charnumber>
  - funnydirnames/<char>
- Files, each containing a line of text indicating which file it is:
  - funnyfilenames/a<char><charnumber>
  - funnyfilenames/<char>
  - plainfilename/<charnumber>
- Symbolic links:
  - named "funnysym/a<charnumber>" to "funnyfilenames/a<char><charnumber>"
  - named "funnysym/a<char><charnumber>" to "plainfilenames/<charnumber>"
  - named "funnysym/<charnumber>" to "funnyfilenames/<char>"
  - named "funnysym/<char>" to "plain filenames/ <charnumber>"
- Hard links:
  - between "funnyfilenames/a<char> <char number>" and "funnyhard/a<charnumber>"
  - between "funnyfilenames/<char>" and "funnyhard/<charnumber>"

Since hard links are symmetrical (the two names are equally applicable to the file), there is no particular reason to believe that strange characters will have a different effect in a hard linked file than in a normal file. Stranger things have happened, but even if there is an effect it's liable to be dependent on which name the program encounters first, which is difficult to control for. The main reason for the hard links is simply to ensure that the test directory has a very large number of hard linked files in it, since that presents problems for some archive programs.

Because Unicode characters are pairs (or triples) of 8-bit characters, I test 8-bit characters in pairs. This test actually runs through both valid Unicode characters and meaningless pairs, since it tests every pair where both halves have the eighth bit set (values from 128-255). It creates the same sorts of directories, files, and links as for the 7-bit characters, but sorts them into subdirectories by the value of the first character in the pair (otherwise, the directories get large enough to be annoyingly slow).

Long Names

Traditionally, UNIX-based operating systems have two kernel parameters that control the maximum length of a filename, and they are "MAXCOMPLEN" and "MAXPATHLEN". MAXPATHLEN is known as PATH_MAX in some kernels, and MAXPATHLEN as a concept has been removed from the Hurd. MAXCOMPLEN is the maximum length of a component in a path (a directory name, or a file name). Traditionally, it's 255 characters in Berkeley-based systems, but your kernel may vary; unless you are running an eccentric system, it is unlikely to be lower than 128 or higher than 512, although some traditionalists may still be running systems with 15 character limits.

MAXPATHLEN is **not** the maximum length of a path. Instead, it is the maximum length of a **relative** path which can be passed to the kernel in a single call. There is no absolute limit on the length of a path in standard UNIX-based operating systems; in any given filesystem, there is a practical limit, which is MAXCOMPLEN times the number of free inodes you have to devote to building a directory tree. In order to build a path over MAXPATHLEN, all that's necessary is to do something like:

```
while (1) {
    mkdir directorynamethatis\
somewherenearmaxcomplenjustfor\
convenience
cd directorynamethatissome\
wherenearmaxcomplenjustforconvenience
}
```

Unfortunately, many programmers have had the understandable but unfortunate impression that MAXPATHLEN is the maximum path length. This leads to any number of undesirable effects (most notably, for a long time fsck simply exited upon encountering such a path, leaving an unclean and unmountable filesystem). Since the first time I ran these tests, this situation has dramatically improved (possibly because a number of evildoers independently discovered this as a user-level denial of service attack against systems). Nonetheless, no backup program I tested will successfully back up and restore paths over MAXPATHLEN.

Saving and restoring files using relative rather than absolute paths would allow backup programs to avoid problems with MAXPATHLEN, at the expense of drastically complicating the programs and the archive formats. It's not clear whether this would be of any benefit; situations where valid data ends up with absolute paths longer than MAXPATHLEN are rare, because people find path names that long inconvenient. However, it's easy to construct scenarios where people using the filesystem as a database could end up with such path names.

First, the test program finds MAXCOMPLEN by experimentation, creating files with names of increasing length until it gets an error. It creates the following:

- Files:
  - "longfilenames/<length>" padded to the length with "a" (as in "longfilenames/10aaaaaaaa")
  - "longfilenames/<length>" padded to the length with newlines
  - "longfilenames/<length>" padded to the length with a valid Unicode character (and an extra "a" if needed, since the Unicode character requires two bytes)
- Symbolic links:
  - named "longsymlinks/<length>" to "long filenames/<length>a..."
  - named "longsymlinks/q<length>" to "long filenames/<length>\n..."
  - named "longsymlinks/u<length>" to "long filenames/<length><Unicode>..."

Then, it builds directories of every length from 2 to MAXCOMPLEN, each of which contains files with names of every length from 2 to MAXCOMPLEN. This may seem silly, but it finds some strange problems that are sensitive to exact path lengths. Each of these files is linked to by a symbolic link with a short name.

Next, it finds MAXPATHLEN by creating directories with names of MAXCOMPLEN, filling each one with names of every length from 2 to MAXCOMPLEN, and descending until it gets an error.

Finally, it ensures that there are paths from the root of the test directory that are over MAXPATHLEN by changing working directories down a level and creating another directory tree out to MAXPATHLEN, this time without filling all the directories out with files. In order to ensure that there are files in the bottom directory, the directory names are slightly shorter than MAXCOMPLEN (due to people's fondness for powers of 2, MAXPATHLEN is generally an even multiple of MAXCOMPLEN). The bottom directory is filled out with filenames until it gets an error.

Access Permissions

Some archive programs have had problems saving or restoring files with tricky permissions. For instance, they may skip files that don't have read permission (even when running as root). Or, they may do straightforwardly silly things like restoring a directory with its permissions before restoring the files in the directory, making it impossible to restore files in a write-protected directory.

The test program creates a file and a directory with each possible combination of standard UNIX permissions (whether or not it makes sense), by simply iterating through each possible value for each position in octal. Each directory contains a file with standard permissions.

Holes

Some filesystems support a spacesaving optimization where blocks that would not contain any data are not actually written to the disk, and instead there is simply a record of the number of empty blocks. This optimization is intended to be transparent to processes that read (or write) the file. On writing, these are blocks that the writing process has skipped with "seek" or its equivalent. A process that reads the file will receive nulls, and has no way of knowing whether those nulls were actually present on the disk, or are in a skipped disk block. Skipped disk blocks are known as "holes," leading to any number of jokes about "holey" files. Files with holes in them are also known as "sparse" files.

This feature is actually used with some frequency; core dumps usually contain holes, as do some database files. These can be extremely large holes, and core dumps often occur on relatively crowded root file systems. Therefore, filling in holes can make it impossible

to restore a filesystem, because the restored filesystem is larger than when it was backed up and may not fit on the same disk. On the other hand, swap files and other database files commonly reserve disk space by genuinely writing nulls to the disk. If these are replaced by holes, you can expect performance problems, at a minimum, and if the space that should have been reserved is used by other files, you may see crashes.

Because a reading process simply sees a stream of nulls, regardless of whether they are on the disk or represented by a hole, it is difficult for backup programs that read files through the filesystem to determine where there are holes. On most modern filesystems, the available information for the file does include not only length (which includes holes), but number of blocks (which does not) and size of a block. By multiplying the number of blocks times the size of a block and comparing it to the length of the file, it's possible to tell whether or not there are holes in the file. If the file also contains blocks that were actually written to disk containing nulls, there's no good way to tell which blocks are holes and which are genuinely there. Fortunately, such files are rare.

The test program creates a large number of files with holes; first, there is a file with a straightforward hole, then a file full of nulls that does not contain holes, and then files with 2 through 512 holes, then a file with a four gigabyte hole, and finally (if the filesystem will allow it, which is rare) a file with a four terabyte hole.

Running the Tests

The basic procedure for running the tests was to use the test program to create a directory, to run that through a backup and restore pair (if possible, by piping the backup straight to the restore), and then to compare the resulting directory with the original, using diff -r to compare content and a small Perl program to compare just the metadata. The backup and restore were both run as root, to give the program a maximum ability to read and write all the files. If special options were needed to preserve permissions and holes, I used them. If there was an available option to do the read and write in the same process, I tested that separately. Usually I tested both the default archive format and the archive format with the support for the largest number of features; I did not test old archive formats that are known not to support the features I was testing except where the program defaulted to those formats. (In at least one case, I wasn't able to test the program default because it simply wasn't capable of backing up any files on the filesystem I was using – I had accidentally tested large inode numbers!)

I ran the tests on the archive programs installed by default on the operating systems I was testing, plus some archive programs I found on the network. The operating systems were chosen relatively unscientifically; I was aiming for availability and popularity, not necessarily quality. Your mileage will almost certainly vary, running programs with the same names, depending on the program version, the operating system version, the type of filesystem you are using, and even the size of the filesystem (large disks may result in inode numbers too large for some archive formats). I tested on RedHat Linux 8.0 with ext2 file systems, FreeBSD 4.8 with ufs file systems, and Solaris 5.9.

The testing program's approach is brute force at its most brutal, and as such it treats "." and "/" as normal characters, even when attempting to make hard links. This results in tests that attempt to hard link directories together, which is flagrantly illegal. Solaris actually allows this operation to succeed, creating an extra challenge for the programs being tested.

In a few cases, I tested other features; incrementals, exclusion of file names, tables of contents, comparison of archives to directories. When people are actually using backup and archive programs, they tend to use these features, which may change results. In particular, if you use an archive program within a wrapper, the wrapper may rely on tables of contents to determine what was archived; if the table of contents doesn't match the archive, that's a problem. Note that if the table of contents is fine, but the wrapper program has difficulties of its own parsing strange characters or long file names, you still have problems; I didn't test any wrapper programs, but given that many tables of contents use newline as a delimiter, and also put unquoted newlines in filenames, I can only assume that problems will result.

The Results

Tar

Classic tar has very clear limits, imposed by the archive format. These limits include:

- No support for holes
- 100 character filename limit
- No support for devices and named pipes

There is a newer POSIX standard for an extended format that fixes these problems, and most versions of "tar" are now capable of using it.

In my original testing, most tested systems were using classic tar; the POSIX standard was sufficiently new so that only a few operating systems were using it, and as a result tar tested very badly. Even the systems that were using newer tar formats tended to have short filename limits. GNU tar did much better.

Both RedHat and FreeBSD ship GNU tar as tar, which defaults to using the extended format.

RedHat: GNU tar 1.13.25

```
tar -cSf - . |
    (cd ../testrestored; tar -xpf -)
```

Removed every symbolic link where the target's last directory component + filename = 490, every file where the last directory component + filename = 494.

Note that longer filenames were handled perfectly, up to MAXPATHLEN. Converted blocks of nulls to holes.

I also tested incrementals (touching every file in the test directory and using tar's list feature), which worked as expected, providing the same results as fulls. Simple exclusion of a test filename worked correctly, both with a normal character and with 8-bit characters. Compressing with -z or -Z provided the same result as testing without compression.

Comparing the archive to the test directory did not work. Tar complained about header problems, complained that the files with holes in them were too large, complained that the archive was incorrect, and finally dumped core.

FreeBSD: GNU tar 1.13.25

I tested only straightforward tar on FreeBSD; the results were identical to the results under RedHat.

Solaris: GNU tar 1.13.19

I tested only straightforward GNU tar on Solaris. Aside from the linking problems mentioned earlier, the only problem was converting blocks of nulls to holes.

Standard Solaris tar

```
tar -cf - . |
    (cd ../testtar; tar -xpf - )
```

Filled in holes. Omitted all files with names over 100 characters long. Omitted all directories with names over 98 characters long. Omitted all symbolic links with targets over 100 characters long.

Dealt with erroneous hardlinked directories by linking the individual files within the directories.

```
tar -cEf - . |
    (cd ../testtare; tar -xpf - )
```

This uses an extended tar format that supports longer names. Unfortunately, it does not appear to support 8-bit characters. It did not succeed in backing up files with 8-bit characters in names or symbolic link targets, and crashed when it processed too many of them, making it difficult to complete the tests correctly. Filled in holes.

Cpio

Cpio relies on find to provide lists of filenames for it, so some of its performance with filenames depends on find's abilities. In the original testing, this presented problems, but since RedHat and FreeBSD both ship GNU cpio and GNU find, which are capable of using nulls instead of newlines to terminate filenames, this problem has been significantly reduced.

Cpio's manual page suggests using the -depth option to find to avoid problems with directory permissions. It does not explain that if you actually do this, you must also use special cpio options to get it to create the directories on restore. I tested both depth-first and breadth-first.

I did not test cpio under Solaris, because of time constraints.

RedHat: GNU cpio 2.4.2

```
find . -true -print0 |
    cpio -0 -o --sparse -H newc |
    (cd ../testcpio; cpio -i)
```

"-H newc" uses newcpio format; this is necessary because as it happens the test filesystem is on a large disk and has large inode numbers, and the default format could not dump any data from it.

Cpio could not handle the larger files with holes in them, claiming they were too large for the archive format, and converted blocks of nulls to holes. (This test run was missing the large file without holes in it.) Otherwise, handled all tests correctly up to MAXPATHLEN.

```
find . -depth -true -print0 |
    cpio -0 -o --sparse -H newc |
    (cd ../testcpio; cpio -i -d)
```

Exactly the same results as without -depth, suggesting that -depth is not in fact important in current versions of cpio if you're running as root.

```
find . -depth -true -print0 |
    cpio -0 --sparse \
        -d -p ../testcpioio
```

Exactly the same results as running a separate cpio to archive and to restore.

FreeBSD: GNU cpio 2.4.2

```
find . -print0 |
    cpio -0 -o --sparse -H newc |
    (cd ../testcpio; cpio -i)
```

The underlying filesystem would not allow creation of the largest files with holes, so those weren't tested. The four gigabyte file was truncated to four blocks, and all holes were filled in. Otherwise, handled all tests correctly up to MAXPATHLEN. Note that although this is the same cpio version tested on RedHat, it did not return the same results! On RedHat, holes were handled correctly.

```
find . -depth -print0 |
    cpio -0 -o --sparse -H newc |
    (cd ../testcpiodepth; cpio -i -d)
```

The same as without the -depth option.

```
find . -depth -print0 |
    cpio -0 --sparse -d -p ~zwicky/testcpioio
```

Omitted paths over 990 characters (MAXPATHLEN is 1024). Correctly handled holes, aside from converting the block of nulls to a hole. Truncated the large file from four gigabytes to four blocks.

Star

Star is an archiver which uses the new extended tar format. It isn't particularly well-known, but it ships with RedHat. It was not tested in my original tests.

RedHat: star 1.5a04

```
star -c -sparse . |
    (cd ../teststar; star -x )
```

Reports 14 files with names too long, one file it cannot stat. Passes all tests up to MAXPATHLEN except that blocks of nulls are converted to holes.

Pax

Pax is an archiver which will use tar or cpio formats; it was one of the first implementations of the POSIX extended tar formats. It doesn't appear to be particularly well-known, despite the fact that it's widely available. Unfortunately, it defaults to the old tar format.

In my original testing, pax did relatively well, correctly handling devices and named pipes, handling holes (but converting blocks of nulls to holes), and correctly handling all file names. It had unfortunate length problems, at different places depending on the format, and bad permissions on directories made it core dump (regardless of format).

RedHat

```
pax -w . |
    (cd ../testrestored; pax -r -p -e)
```

Removed every file with filename at or over 100 characters, every symbolic link with target at or over 100 characters, every directory with a name at or over 139 characters. Turned blocks of nulls into holes.

```
pax -w -x cpio . |
    (cd ../testpaxcpio; pax -r -p -e)
```

Truncated all symbolic link targets to 3072 characters (this is 3 * 1024, which is probably not a coincidence). Returned error messages saying that filenames with length 4095 and 4096 are too long for the format, but in fact it also fails to archive all paths over 3072 characters. Blocks of nulls become holes. The largest files are missing, with a correct error message.

```
pax -r -w -p e . ../testpax3
```

The same results as with cpio format, except that large files are now present.

FreeBSD

MAXPATHLEN on ufs is only 1024 characters, and the large files with holes are not created, so there are no problems with symbolic link targets, long paths, or large holes in files. Otherwise, the results are identical to the results on RedHat; the net effect is that using read-write or cpio mode, all tests are passed up to MAXPATHLEN, except that blocks of nulls are converted to holes.

Solaris

```
pax -w . |
    (cd ../testpax2; pax -r -p e)
```

Removed every file with a filename over 100 characters. Pax then exited after the long filename tests; long hard links, pathnames, and symbolic links were not tested. Changed blocks of nulls to holes; shrank holes (all files with holes larger than one block in them still had holes but were larger on disk than in the original).

Dealt with the erroneous hard-linked directories by unlinking them. However, the second copy of the file encountered became 0 length.

```
pax -w -x cpio . |
    (cd ../testpaxcpio; pax -r -p e)
```

Changed blocks of nulls to holes; shrank holes. Otherwise, passed all tests up to MAXPATHLEN; however, MAXPATHLEN was 1024, not long enough to reveal the length problems shown on Linux.

Dealt with the erroneous hard-linked directories by unlinking them. However, the second copy of the file encountered became 0 length.

```
pax -r -w -p e . ../testpaxrw
```

Pax again crashed, this time during the long pathname tests. This time, I repeated the test on the remaining directories with success. Permissions were not preserved.

Dealt with the erroneous hard-linked directories by linking the individual files within the directories. Blocks of nulls were changed to holes, but all holes were transferred correctly.

Dump/Restore

Dump and restore performed almost perfectly in the original testing, only showing problems with filenames at or near MAXPATHLEN.

RedHat: dump and restore 0.4b28

```
/sbin/dump -0 -f ./dump -A \
    ./dumptoc /dev/hda2
/sbin/restore -if ../dump
```

The symbolic link tests were dramatic failures; several long symbolic links (3 of the 15 links where the filename component was 236 characters long) had content from other files appended to the symbolic link target. The most boring of these changed "236aaa...aaa" to "236aaaa...aaaaized", while a more dramatic one finishes up its a's and continues on with "bleXML\verbatimXML readfile{\truefilename{#1} {}{}\endgraf" and so on for another screenfull. This was so peculiar that I repeated the test, with the test directory on a different filesystem; on the second try, only two of the links where the filename component was 236 characters long were affected, and the filesystem had fewer files with interesting content, so the additional text was boring test content from other files in the test directory.

Aside from that, dump/restore passed all tests up to MAXPATHLEN and appeared to produce a correct table of contents. Running its test of the archive produced the unedifying and incorrect result "Some files were modified!" Even if this had been correct, there is no way that it could ever be useful, since it does not tell you which files it believes were modified.

FreeBSD

Passed all tests up to MAXPATHLEN.

Solaris

Restore crashed and dumped core when attempting to add the files that were over MAXPATHLEN to the restore list. Aside from that, passed all tests up to MAXPATHLEN.

Dealt with the erroneous hardlinked directories by restoring the second link encountered as a file.

Dar and Rat

Dar and rat are Linux archive programs I picked up to see whether the improvements in test results were fundamental changes, or just reflected the increased age and stability of the programs under test. It would appear that the latter is the case.

In general, I test archive programs by running them in the test directory, and archiving the current directory ".". (This gives the most possible pathlength.) Neither dar nor rat would archive "." successfully, and neither one would run with an archive piped to a restore.

I did not end up testing rat, due to its habit of exiting the writing process before actually writing any data when it hit files over MAXPATHLEN. Although this is certainly one approach to error handling (it ensures that you are never surprised by an archive that appears good but is missing files), I didn't have time to weed the testing directory down until I could actually get some testing done.

Dar did somewhat better. On the first attempt, when it reached the largest file with holes, it increased memory usage until the operating system killed the process for lack of memory (unfortunately, at this point my entire login session was also dead, but that could be considered RedHat's fault.) Most annoyingly, although this happens to be one of the last files to be written, so that there was quite a large archive file at that point, no data was recoverable from the archive file (apparently it has to finish in order to be willing to read any data from the archive). In this case, since there was one clearly identifiable offending file, I was able to remove it and re-run the tests, and while dar filled in holes, it otherwise passed all tests up to MAXPATHLEN.

Cross-compatibility

Given that a large number of programs support the same formats, it's reasonable to be curious about compatibility between them. There are far too many possible combinations to test exhaustively, but I tested a few to see what would happen. All of these tests were on RedHat.

Star to Tar

```
star -c -sparse . |
    (cd ../teststar; tar -xp)
```

Removed every symbolic link where the target's last directory component + filename = 490, every file where the last directory component + filename = 494. Note that longer filenames were handled perfectly, up to MAXPATHLEN. Many of the files with holes were deleted; in fact, every other file in the archive. There were error messages complaining about header problems. It is impossible to tell whether blocks of nulls were converted to holes, since that file is one of the missing ones, but it seems safe to assume that they would have been.

Tar to Star

```
tar -cSf - . |
    (cd ../testtartostar; star -xp)
```

Correctly backed up all files up to MAXPATHLEN, except that the block of nulls was converted to a hole.

Conclusions

Archive and backup programs have considerably improved in the last 10 years, apparently due to increased maturity in the programs being distributed. There are still unfortunate and peculiar problems, but compared to the frequent core dumps in the previous testing, they are relatively mild.

It is important to note that you cannot draw conclusions about programs based on their names. As the results show, it's simply inappropriate to make generalizations like "dump is better than tar." This is true on Solaris (where the dump is a reasonably mature version, but tar is merely elderly) but false on RedHat (where the dump has not yet reached version 1, but the tar is the reasonably solid GNU tar). In fact, there were significantly different results for GNU cpio running exactly the same version on exactly the same hardware, using different operating systems and filesystem types. It is simply not safe to generalize about the behavior of programs without looking at how the particular version you are using works with the particular operating system, filesystem, and usage pattern at your site.

For the best results in backups and archiving:

- Use a mature program. Treat people who write their own archive programs like people who write their own encryption algorithms; they might have come up with an exciting new advance, but it's extremely unlikely.
- Test the program you are using in exactly the configuration you need it to run in. Apparently minor changes in configuration may cause large changes in behavior.
- Avoid directories at or near MAXPATHLEN.
- Be sure to use options to handle sparse (holey) files, but be aware that this may create extra holes.

Availability

The test programs were written in a spirit of experimentation, rather than with the intention of producing software for other people to use. I strongly encourage people who are interested in testing backup and archive programs to produce their own tests that cover the cases they are most interested in. However,

if you insist on using my programs, or just want to snicker at my programming, they are available from http://www.greatcircle.com/~zwicky

Biography

Elizabeth Zwicky is a consultant with Great Circle Associates, and one of the authors of "Building Internet Firewalls," published by O'Reilly and Associates. She has been involved in system administration for an embarrassingly long time, and is finding it increasingly difficult to avoid saying horrible things like "Why, I remember when we could run an entire university department on a machine a tenth the size of that laptop." Reach her electronically at zwicky@greatcircle.com.

References

[1] Shumway, Steve, "Issues in On-Line Backup," *Proceedings of the Fifth LISA Conference*, USENIX, 1991,.

[2] Zwicky, Elizabeth, "Torture-testing Backup and Archive Programs: Things You Ought to Know But Probably Would Rather Not," *Proceedings of the Fifth LISA Conference*, USENIX, 1991.

An Analysis of Database-Driven Mail Servers

Nick Elprin and Bryan Parno – Harvard College

ABSTRACT

This paper compares the performance of three different IMAP servers, each of which uses a different storage mechanism: Cyrus uses a database built on BerkeleyDB, Courier-IMAP uses maildirs, and UW-IMAP uses mbox files. We also use a mySQL database to simulate a relational-database-driven IMAP server. We find that Cyrus and mySQL outperform UW and Courier in most tests, often dramatically beating Courier. Cyrus is particularly efficient at scan operations such as retrieving headers, and it also does particularly well on searches on header fields. UW and Cyrus perform similarly on full-text searches, although Cyrus seems to scale slightly better as the size of the mailbox grows. mySQL excels at full-text searches and header retrieval, but it performs poorly when deleting messages. In general, we believe that a database system offers better email storage facilities than traditional file systems.

Introduction

Most IMAP and POP3 servers, even ones commonly used in environments with many users, store mail data in flat text files. This suffices when the quantity of email is small, but when an email application must sift through hundreds of megabytes of inefficiently stored data, server performance can suffer dramatically. A recent analysis of a typical university file system found "a population of users who spend the majority of their file system accesses reading email" [1]. The paper concludes by noting: "While it may have been obvious a priori, flat-file mailboxes are quite inefficient. Anecdotal evidence suggests that database-driven mail servers are faster and consume fewer resources than file-system based ones, and we have no reason to dispute this."

Despite the alleged a priori obviousness of the benefits of a database-driven mail storage system – or perhaps because of it – there has been little empirical exploration of the potential for database use in mail servers. Yet the problem of email storage seems perfectly suited for a database solution [2]. Databases allow for more advanced and efficient content queries (particularly sorting and searching data) than flat files and, moreover, a cleanly structured database schema for email would better mediate the transfer and exchange of mail data to other applications and formats.

Background And Related Work

To our knowledge, little work exists comparing the performance of traditional file-based email servers with that of a database-driven system, although several papers do propose and defend various types of file-based email systems. Sam Varshavchik, for example, compares mbox with maildir mail storage [3]. The mbox system concatenates every message together and stores the result in a single file, while the maildir format stores each message in a separate file. Even though Varshavchik concludes that maildirs will match or outperform mbox systems in all but a few isolated cases, he admits that maildirs still contain inherent flaws. Among them, he mentions that maildirs will perform poorly if used for specific content searches on large folders, particularly if the server runs on a machine with an inefficient file system.

Mark Crispin also highlights the shortcomings of file-based mail storage [4]. He notes that most operating systems synchronize file creation. As a result, any attempt to create or delete email runs into a narrow bottleneck at the level of the file system. Furthermore, a text search requires opening and closing every file in the mail directory, placing a considerable strain on the file system. In the end, Crispin concludes that a database-driven solution would result in superior performance, but he offers no arguments to support his position, other than his comments on the shortcomings of file-based systems.

Several companies offer email server products based on database storage. DBMail offers programs that "enable the possibility of storing and retrieving mail messages from a database" [5]. The site claims the system performs faster queries, scales better, and provides better flexibility than file-based storage, but it offers little reasoning or evidence to support these claims.

Openwave Email Mx (formerly Intermail), a product from Software.com, uses Berkeley DB, an embedded data store, to archive messages. But again, aside from its claim that Email MX sets "new standards of excellence with its massive scalability, unrivaled performance, and superior architecture," the company provides no information on the server's performance compared to traditional mail systems [6]. Similarly, Citadel also bases its mail storage on a database, but offers no evidence to support this decision [7]. In fact, the only argument advanced in

favor of using a database relies on creating a single instance of each email that enters the system, so that if multiple users receive the same email, the system stores only one copy. Without evidence to support the conjectured performance improvement, we learn little about the relative merits of database versus file-system storage.

Thus, although researchers have argued over the advantages and disadvantages of different file-based mail storage systems, they have neglected the subject of database-driven solutions. A few companies do offer such solutions, but as far as we could find, none provides any evidence to justify the alleged benefits of this alternative.

Experimental Setup

We tested IMAP servers with three different storage mechanisms. Courier IMAP (v. 1.4.2) uses maildirs, UW-IMAP-2002 uses mbox files, and Cyrus (v. 2.1.9) uses BerkeleyDB; additionally, we created an email database in mySQL (v. 3.23.49) and used it to model a relational-database-driven IMAP server. Other mailbox formats, such as the more performance-oriented mbx format, do exist, but we believe the popularity of mbox and maildirs justifies their use in these tests. And although other database-driven servers also exist – the few noted earlier, as well as more widely known applications from Microsoft and Sun Microsystems – we chose to include only non-commercial servers in our comparison. All servers were run on a machine running NetBSD 1.6 with a 200 Mhz Pentium processor, 128 MB of RAM, and a 1.5 GB IDE hard drive.

Our benchmark consists of seven basic requests, and we perform each on three different mailbox sizes:[1]

Test Mailbox Sizes			
	Small	Medium	Large
Size	5 MB	30 MB	100 MB
Number Messages	1,046	8,734	21,282

All users have a single folder named 'INBOX'.

Our setup attempts to model email usage and configuration of users from three different categories. The small user represents a typical webmail user with a relatively small quota. The medium user represents the average size for a commercial or academic email account. The large mailbox typifies a high-volume account which regularly accumulates thousands of emails, such as a corporate account. Our basic configuration models the typical naïve user who saves every message in the inbox.

Our mySQL database schema consists of a table of messages for each account size, small, medium, and large. Each table stores header information and the message body in separate fields. We strategically defined indices on properties we expected to be particularly relevant to typical IMAP usage. For instance, we created a full-text index on subject, body, sender, and recipient. Thus the mySQL model and Cyrus used storage mechanisms designed to structure data intelligently, while UW and Courier used standard file system structures. Arguably, the playing field would have been more level with a standard IMAP server altered to use a mySQL database rather than a database model independent of the rest of the IMAP server. We chose the latter option for a number of reasons: we wanted to test publicly available software rather than customized versions; we believe overhead of accessing the data source will outweigh other factors necessary to run an IMAP server; and our time and resource constraints made it difficult to integrate mySQL storage into an existing IMAP server.

To generate a large collection of diverse email, we created an email alias (namely emagnet@hein.eecs.harvard.edu) to the three accounts on each server and then subscribed emagnet to a large variety of mailing lists. Thus, each account received exactly the same email. This approach allowed us to measure each server's performance on real email, rather than artificially created messages. It also guaranteed diversity in the mail headers, so that we could accurately test the servers' searching abilities.

The benchmark times the following requests:

1. *Retrieve all headers in the user's INBOX.*
   All mail clients must routinely perform this operation to download new mail and refresh cached information, so the server's response should be quick and efficient. This procedure is analogous to a scan operation that returns a single field's value for every tuple in a relation.
2. 2a. *Retrieve a list of messages whose full text (subject and body) contains "happy" (chosen to match approximately 1 percent of the messages).*
   2b. *Retrieve a list of messages whose full text contains "free" (chosen to match approximately 25 percent of the messages).*
   The IMAP protocol allows users to retrieve mail that matches conditions a user can specify in a simple query. Performance of this operation must scale well with the size of the mail data and the number of successful matches.
3. 3a. *Retrieve a list of messages whose sender is "parno@fas.harvard.edu"*
   3b. *Retrieve a list of messages received on November 30, 2002*
   Searches on the header fields may be faster if the mail server does not need to open an entire message file to perform the search. A storage system that allows selective access to only those parts of the message that the search requires should outperform a flat-file system on tests like this.

[1] In addition to the three user accounts described above, we also tested a tiny account consisting of 105 messages (0.5 MB). In virtually every test, the results paralleled those of our other three accounts. However, the time differences between servers proved trivial, so we chose to omit them from our results for the sake of brevity.

4. 4a. *Expunge 1 percent of the user's mail.*
   4b. *Expunge 20 percent of the user's mail.*
   Deleting mail is routine maintenance that all email users must perform regularly. This test measures the time necessary to fully purge all mail flagged for deletion. The flagging of messages for deletion was done randomly.

We measured times from the perspective of the mail client application. That is, the time necessary for the request is the difference between the time at which the client sends the request and the time at which it receives data back from the server. We ran each of our tests cold, i.e., the server had not previously accessed the data we queried. To avoid network-related timing delays, our benchmark client runs on the server. We ran each test (except expunging deleted messages) three times; results are the average of the three numbers.

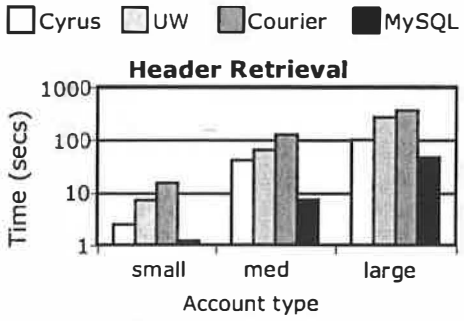
Results

Header Retrieval Times

First, the header retrieval times in tabular format:

Header Retrieval Times (in seconds)			
	Small	**Medium**	**Large**
Cyrus	2.36	39.3	98.2
UW	6.83	61.9	263.7
Courier	14.37	119.4	356.7
mySQL	1.16	7.08	44.79

All of the standard deviations were less than 1.0. More graphically, this looks like:



For this benchmark, a retrieval of all message headers in a user's inbox, both database-solutions outperform the file-based servers, with the performance differential increasing as the size of the mailbox grows. An intelligent index on the header field makes this a simple sequential scan for both Cyrus and mySQL. Furthermore, our SQL schema gives mySQL an advantage over Cyrus: mySQL isolates each user's messages into a unique table, while Cyrus must scan through all messages to select only those from the account requested. In contrast to both of these, UW must search through the entire mbox file. Worse still, Courier must open and close hundreds – thousands for medium and large – of files and then scan through each file to find the relevant information.

Full-Text Searches

Figure 1 show the tabular and graphic results of searching on "happy" and "free." All of the standard deviations less than 0.3 except as noted: † standard deviations less than 1; ‡ standard deviation = 26.8.

Search on "happy" (times in seconds)			
	Small	**Medium**	**Large**
Cyrus	4.04	25.64	65.6†
UW	2.92	17.76	64.22
Courier	7.76†	52.2†	209.1‡
mySQL	0.71	5.49	15.28

Search on "free" (times in seconds)			
	Small	**Medium**	**Large**
Cyrus	3.81	24.55	62.4†
UW	2.79	17.43	64.19
Courier	6.56†	50.5†	193.6†
mySQL	1.62	11.32	35.77

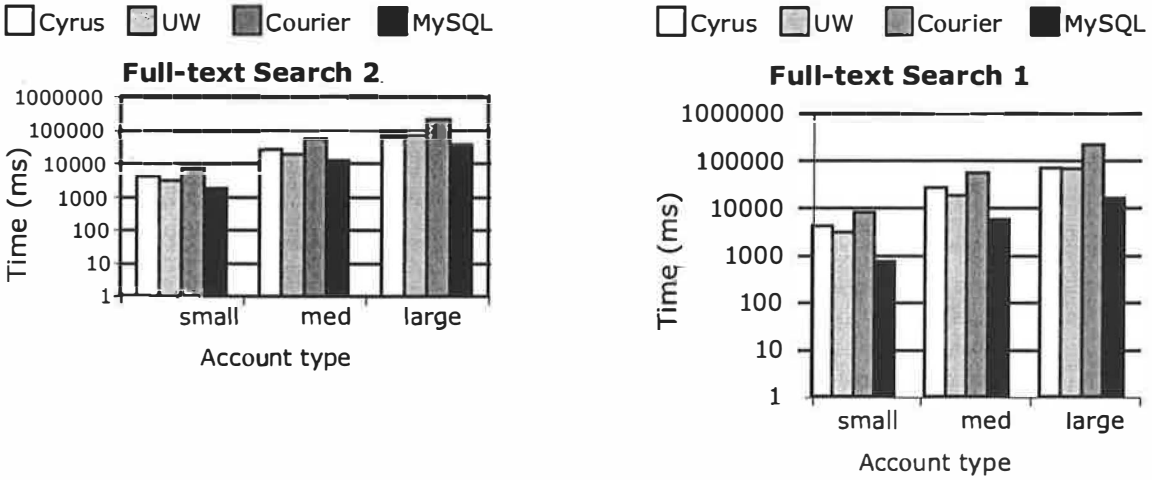


Figure 1: Retrieval times for searching on "happy" and "free".

This test measures the speed of a full-text search on messages' subjects and bodies. On the small and medium accounts, UW responds slightly faster than Cyrus, though both outperform Courier by a significant margin. Cyrus's somewhat disappointing statistics may result from the fact that it must first isolate the specific user's mail from all stored messages, and only then can it perform a full-text search. Meanwhile, UW performs a relatively simple text search through a single file. Simple file searches such as these can often outperform databases by avoiding the extra overhead dictated by a DBMS. As the account size grows however, Cyrus's database scales better than UW's flat files: Cyrus's times on the large-account search are faster than, or about equal to, UW's times.

Beating UW and Courier by a much wider margin, mySQL clearly demonstrates the advantage of intelligent indexing. Although it takes initial overhead to precompute[2], the full-text index allows mySQL to find text in message bodies much more quickly than all three other solutions. Overall, this suggests that a database solution will be most efficient when indexing is utilized or a user's mailbox constitutes a majority of the total mail stored on the server. The disadvantage of a database solution is that the combined size of all accounts affects its performance on each account, so the worst-case scenario is when a user's account is small but the server stores a large quantity of mail in other accounts.

[2]One could argue that the cost of maintaining a full-text index would also slow the speed with which the server could store incoming messages. Even if the cost of indexing a single message were significant, however, it would not be relevant to our benchmarks, which time queries from the perspective of the end user. It is unlikely that indexing incoming messages would affect the perceived speed of a user trying to search messages, for example.

Searching Specific Header Fields

See Figure 2 for the graphical results of searching the 'From' and 'Date' header fields. All standard deviations were less than 0.1, except as noted: † standard deviation = 3.6; ‡ standard deviation = 0.93 .

On a search over specific header fields, Cyrus and mySQL once again outperform both file-based servers on all three accounts, with the exception of UW's superior performance on the date search on the small account. Given that the database solutions still scale in a far superior manner, this exception seems negligible. An index on the header information makes this search an extremely simple database problem, though without the overhead of a traditional relational database, Cyrus can execute these queries more quickly than mySQL. UW, while still slower than Cyrus, outperforms Courier – UW merely performs the equivalent of a grep on a single file, while Courier must open and close every single message file.

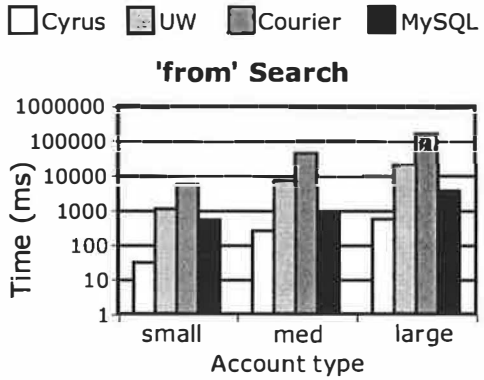
Expunging

Figure 3 shows the times for expunging 1 and 20 percent of the box's mail.

Expunging messages flagged for deletion proved to be a particularly elucidating test. Cyrus dramatically beats UW when purging a relatively small number of messages but takes much longer than UW when the number of deleted messages grows to 20 percent. Opening the mbox file and rewriting it are the most costly disk operations UW must perform; identifying messages for deletion once the file is open should be relatively quick, and the more messages UW deletes, the less data it must write back to disk. In contrast, although Cyrus seems to have quicker access to the messages it must delete, it scales more slowly when it

Searching 'from' Header (times in seconds)			
	Small	**Medium**	**Large**
Cyrus	0.03	0.25	0.55
UW	1.02	6.67	20.33†
Courier	5.37	45.4	164.9‡
mySQL	0.52	0.85	3.51

Searching 'date' Header (times in seconds)			
	Small	**Medium**	**Large**
Cyrus	0.06	0.18	0.36
UW	0.026	0.39	1.21
Courier	0.35	3.78	56.1
mySQL	0.27	0.32	0.8



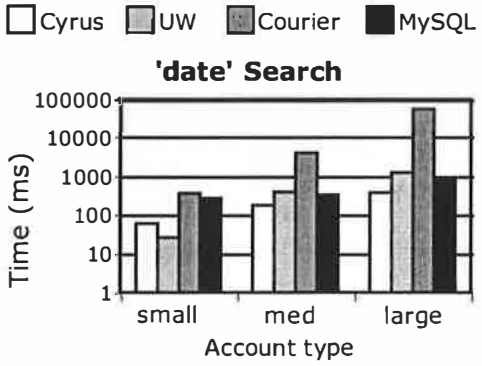


Figure 2: Timing for searching header fields.

deletes a large number of messages. Cyrus must also update its indices as it deletes, and this can be a costly operation for a deletion as large as 20 percent. Data for mySQL further demonstrates the cost of maintaining extensive indices: although our SQL schema allows mySQL to perform queries quickly, its performance suffers severely when it must delete messages. Courier's deletion times scale approximately linearly with the number of messages. Courier's deletion process requires no significant overhead, it is simply a matter of deleting one file for each message.

General Analysis

In general, the database solutions, Cyrus and mySQL, consistently outperform the file-based servers in both response time and scalability, often by orders of magnitude. Moreover, Cyrus and mySQL have buffer caches that enable even faster performance than our results indicate.[3] Executing queries on the same data sequentially yielded significantly faster results for both Cyrus and mySQL. For example, although Cyrus requires four seconds to perform the first-full-text search, running a second full-text search immediately after the first one requires only one second. Similarly, a search on recently queried data in mySQL often takes less than a tenth of a second.

For Cyrus, warm runs only yielded faster times on the full-text searches. It presumably has indices on header fields, so caching records would not help it perform searches on these data. Warm runs yielded a similar performance increase on the medium account but had no effect on Cyrus's large account, probably for the simple reason that the large account has too much data to cache. MySQL, in contrast, exhibited significant performance improvements on all accounts and all queries when using warm data. UW-IMAP and Courier's times remained unchanged on warm runs, however, so clearly caching endows database implementations with a significant advantage over file-based storage mechanisms in certain contexts.

Compared with UW and Courier, Cyrus does perform poorly on our twenty-percent deletion test, and mySQL performs embarrassingly on both deletions. We do not believe this is a significant problem, however, as users rarely delete such large quantities of mail at once; instead, we suspect the one-percent deletion occurs far more regularly. Furthermore, deleting a large amount of email can happen, by nature, only infrequently, while header retrieval takes place constantly, so it is far more important that a mail server handle the latter task efficiently.

Although mySQL outperforms the other solutions in header retrieval and full-text searches, it has the advantage of avoiding the overhead of an actual IMAP server implementation. For example, mySQL avoids network connections, message-wrapper data structures, and IMAP command parsing. In general, however, the majority of the time processing each benchmark consists of executing the query rather than performing quick operations like parsing the IMAP commands. Furthermore, since none of our benchmarks measured the time needed to connect to the servers, adding an actual IMAP server on top of the mySQL database would not add to the overhead of a connection. Thus, our simulation presents a reasonable approximation of a relational-database solution.

[3]Client-side caching is also important for performance from the end-user's point of view, but it is not germane to our current discussion. Please see Varshavchik's analysis [3] for a more thorough discussion of performance and client-side caching.

Expunge 1% of mail (times in seconds)			
	Small	**Medium**	**Large**
Cyrus	0.68	4.52	12.13
UW	0.82	6.14	53.69
Courier	0.41	4.72	13.69
mySQL	2.84	29.47	120.56

Expunge 20% of mail (times in seconds)			
	Small	**Medium**	**Large**
Cyrus	6.17	64.63	156.50
UW	0.93	8.25	69.25
Courier	15.18	61.83	195.76
mySQL	7.71	155.14	1086.39

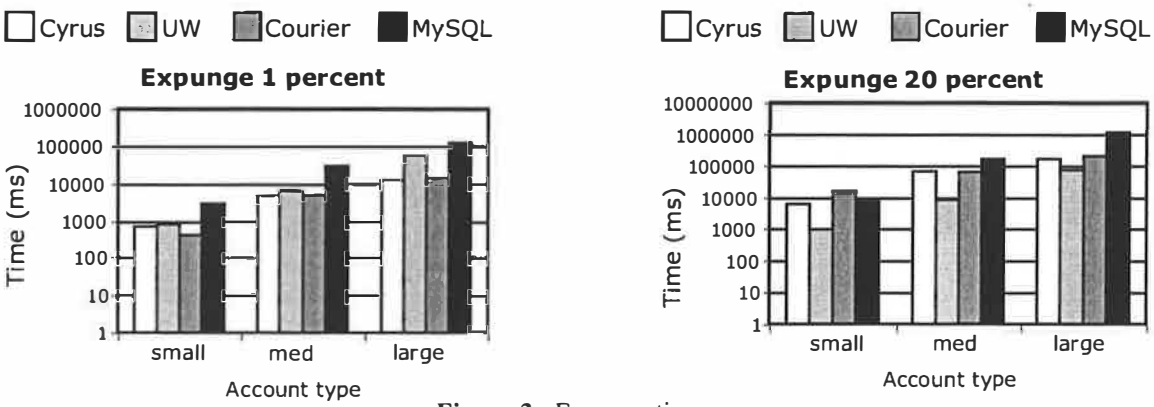


Figure 3: Expunge times.

Our results differed dramatically from Varshavchik's [3], the only other empirical data we have found. Varshavchik concludes that "using maildirs will be just as fast – and in sometimes [sic] much faster – than mbox files," while our results indicate that Courier's maildirs scale terribly compared to the storage mechanisms of Cyrus and UW. We believe at least two factors contribute to this discrepancy. First, according to Varshavchik, "Maildirs will not scale very well on servers that use old, slow, hardware." Indeed, his "low-end" configuration, on which UW did outperform Courier, is similar to our hardware configuration. Also, we tested accounts with both more messages and a greater aggregate size than the accounts in Varshavchik's benchmarks. Instead of upper limits of 10,000 messages and 40 MB, we tested accounts up to over 20,000 messages and 100 MB. Similarly, Varshavchik's benchmarks automatically generated batches of homogeneous messages while we used real email from an eclectic mix of sources; our email contained heterogeneous content and significant variance among message sizes.

Areas For Future Research

Varying some of the benchmark parameters could yield additional interesting and illustrative results. For example, no sane email user would keep 21,000 messages in one folder, so it would be more realistic to run tests with mail distributed among multiple folders. Since the IMAP protocol allows a user to work with only one folder at a time, we felt the results from such tests would not differ significantly from our other results, but this is certainly open to further investigation. In general, more advanced queries, such as a search over multiple header fields, would also offer room for more exploration.

With fewer hardware limitations it would be much easier to run our benchmarks on scales that more accurately resemble large corporate and academic computing environments. It would be interesting, for example, to see how well the different storage systems perform when the server has hundreds of users and must store over a gigabyte of email. Similarly, an investigation of server performance during concurrent access from multiple clients could reveal other advantages of a database storage system; unfortunately, limitations in our experimental setup prevented us from pursuing this inquiry.

Additionally, although our study examines email server performance from the client's perspective, a system administrator may care more about the memory, processor time or disk I/O usage of the mail server, all of which may vary based on the storage system selected.

Finally, we believe that database storage systems will allow mail servers to evolve more advanced features, and an analytic investigation of these possibilities could be a fruitful endeavor. For example, advanced queries could allow for server-side junk mail filtering. More generally, client email applications can be more compact and efficient if they can rely on the server for additional mail-management features, and this could stimulate the development of truly practical email clients on small mobile devices such as cellular phones.

Conclusion

Many features of a DBMS are highly advantageous from the point of view of an IMAP server. The obvious performance differential between the database options and both UW and Courier indicates that email storage is indeed a problem well suited for a database solution. Indexing capabilities give Cyrus and mySQL an advantage over Courier and UW when scanning headers and searching header fields. mySQL's full-text index provides a particularly expedient method for searching through message text, although it adds significant maintenance cost to operations such as adding and removing messages. A server-side buffer cache also improves performance by speeding up searches on recently accessed data. Although UW outperforms Cyrus by a small margin on some full-text searches, mySQL demonstrates clearly that a DBMS can search email much more quickly than a file-based solution. Most importantly, these results offer desperately needed empirical data comparing the performance of these three storage implementations.

Acknowledgements

We would like to thank David Holland and Noam Zeilberger for their technical assistance and UNIX troubleshooting, as well as Stuart Schechter for suggesting this line of research. Dan Ellard and Margo Seltzer were both indispensable for their thoughtful criticism and willingness to help.

Author Information

Nick Elprin (elprin@fas.harvard.edu) is a third-year computer science undergraduate at Harvard College. Bryan Parno (parno@fas.harvard.edu) is a fourth-year computer science undergraduate at Harvard College.

References

[1] Ellard, Daniel, Jonathan Ledlie, Pia Malkani, and Margo Seltzer, "Passive NFS Tracing of Email and Research Workloads," *Second Annual USENIX File and Storage Technologies Conference*, pp. 203-216, San Francisco, CA, March, 2003.

[2] Silberschatz, Avi and Stan Zdonik, et al., "Strategic Directions in Database Systems-Breaking Out of the Box," *ACM Computing Surveys*, Vol. 28, Num. 4, pp. 764-778, December, 1996.

[3] Varshavchik, S., http://courier-mta.org/mbox-vs-maildir/ , 2001.
[4] Crispin, M., *Mailbox Format Characteristics*, http://washington.edu/imap/documentation/formats.txt.html , 1999.
[5] *DBMail*, http://dbmail.org .
[6] *Openwave Email Mx*, http://openwave.com .
[7] *Citadel*, http://uncensored.citadel.org/citadel/ .

A Secure and Transparent Firewall Web Proxy

Roger Crandell, James Clifford, and Alexander Kent – Los Alamos National Laboratory

ABSTRACT

The LANL transparent web proxy lets authorized external users originating from the Internet to securely access internal intranet web content and applications normally blocked by a firewall. Unauthenticated access is still, of course, denied. The proxy is transparent in that no changes to browsers, user interaction, or intranet web servers are necessary. The proxy, a few thousand lines of C running on Linux, has been operating within Los Alamos National Laboratory's firewall since 1999.

Introduction

Our goal was to provide a transparent firewall web proxy, a proxy allowing authorized and secure access to the many firewall-protected intranet web servers at Los Alamos National Laboratory (LANL). Our offsite users required access to internal news, phone directories, web email, the online library, and administrative services. Additionally, they need to download documentation and software. External users often find themselves in the situation where they are forced to use available computers at the sites they are visiting. Non-standard access methods, including browser configuration modification or installing virtual private networking (VPN) software was often impractical or undesirable. At the same time, it was very important to maintain the security of our intranet servers and their data. Only requests from authorized users would be forwarded and all information sent outside the firewall had to be encrypted. We wanted the web proxy to function within the framework of the existing firewall.

We considered several approaches.

- We could make exclusive use of virtual private networking to access internal web servers.
- A proxy could be designed to work only for a single web server, making the design easier.
- Selected web services could be relocated outside the firewall.
- Rules to allow direct access through the firewall to internal web servers could be allowed. These rules could be permanent or dynamic in nature. Dynamic rules would be installed automatically following authentication and removed after a specific time period. Rules would be based upon external IP address and internal web server IP address.
- We could not allow external access to our internal web servers.
- A transparent web proxy could be provided.

We concluded a transparent web proxy would meet all our requirements. The next question was how to implement it in a secure fashion while still providing simple and reliable operation.

This paper focuses on how these needs were addressed through the design of an innovative and user-friendly proxy using available technologies and open source software.

Goals

Controlled access to intranet web servers from the Internet was the goal. VPN access into the intranet was available, but only provided access to users who had the proper VPN client software installed. In addition, a VPN solution was seen as overkill for those users requiring just web-based intranet access. Clearly, there was an accessibility gap needing to be filled. What was needed was an access mechanism that worked with any browser, anywhere. We desired simplified access while also ensuring client transparency, encrypting all data transmitted over the Internet, providing user-based authentication, and requiring no modifications to existing intranet web servers. The authentication method must continue to work in the same manner as the existing firewall. A web single sign on was determined to be a requirement. Once an individual authenticated to the web firewall proxy, they would not need to authenticate again to internal web servers. The authentication on the web proxy would provide authentication credentials necessary to access all internal servers.

Design

Meeting the transparency requirement was our first task. Accomplishing this requires all inbound intranet web requests to traverse a path allowing evaluation of the connection and forwarding as necessary. This evaluation could be done with a broadcast network firewall DMZ where all the incoming HTTP requests would be visible. The proxy could then intercept the requests from the broadcast medium. Another option is to use a stateful Linux firewall [14] which would also act as the proxy. However, we chose a third option: using a router with policy routing [18]. The policy routing rules route packets based upon destination port numbers and sends all HTTP (port 80) and HTTPS (port 443) requests to the proxy server.

The standalone proxy server makes use of the Linux operating system with netfilter [14] and iptables rule set. Through the insertion of iptables rules such as:

```
iptables -t nat -A PREROUTING \
  -ieth0 -p tcp -d INTERNAL_NET \
  -dport 80 -j REDIRECT -to-ports 80
iptables -t nat -A PREROUTING \
  -ieth0 -p tcp -d  INTERNAL_NET \
  -dport 443 -j REDIRECT -to-ports 443
```

the proxy server can accept connections for an entire network and respond using the IP address of the intended destination server. Several router vendors also provide layer-4-based policy routing that provides similar transparent redirection. This policy routing is commonly used for outbound transparent HTTP proxying [4].

The second task was to force the use of SSL encryption on all incoming HTTP connections. This is accomplished by running a redirector daemon on the proxy server. The redirector listens on port 80 and redirects port 80 connection requests to port 443 on the same requested server. To accommodate the fact that the proxy masquerades as many web servers, a wildcard X.509 [13] certificate is installed on the proxy. The certificate has the form *.lanl.gov, Los Alamos' domain name. This simple redirection from HTTP to HTTPS guarantees all data requests from our external users are encrypted.

Once the port 80 to 443 redirect has been issued, knowledge of the original port requested is lost. All web requests arrive at the proxy on port 443, even the ones that started out on port 80. The proxy must make a decision about which intranet web server port it should connect to following authentication. Keeping state information about previous requests was deemed to be problematic, and a simpler approach was needed. It was decided to first attempt a port 443 connection to the intranet server, and look at the HTTP return code [7]. If the return code was 404, indicating the content is not found, the proxy then attempts a connection to port 80 on the intranet server. If the return code is again 404, this is returned to the web browser. If the return code is not 404, the proxy forwards the server's data to the web browser. Figures 1 and 2 indicate the different possibilities described above. Obviously, the proxy does not support intranet web servers running on non-standard ports.

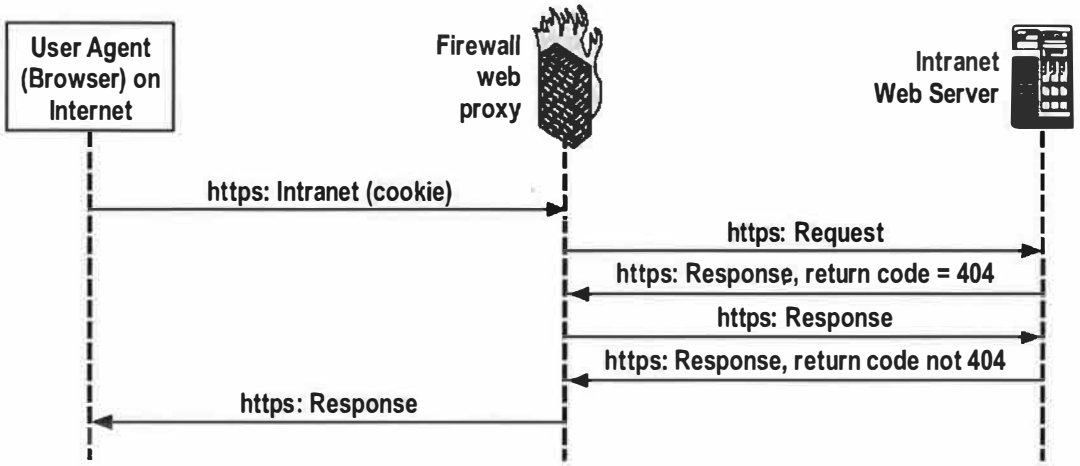


Figure 1: An http request with valid authentication cookie, internal server content on port 80.

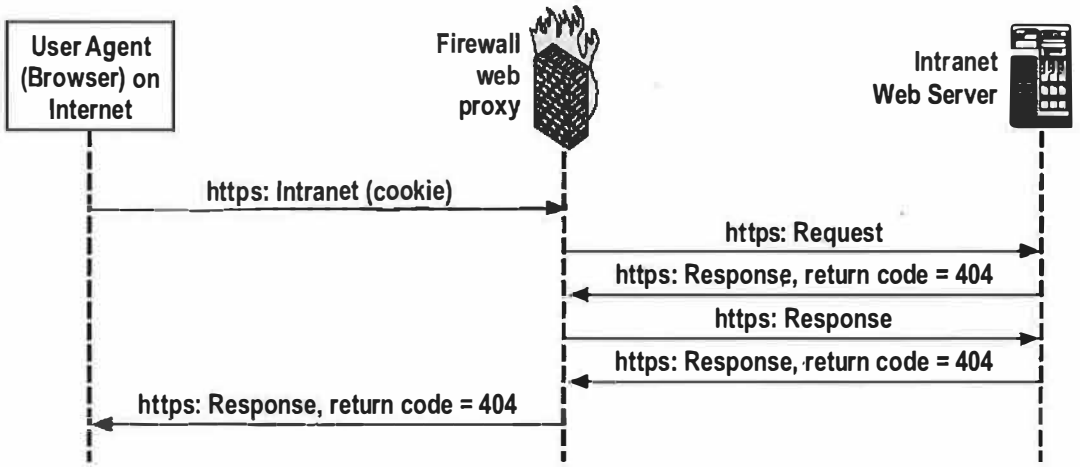


Figure 2: An http request with valid authentication cookie, content unavailable on internal server.

Third, the proxy needs a web authentication system to tell whether or not to forward the user's HTTPS requests. One method we have seen is to remember authenticated users by source IP address. The drawback to this method is you allow access to an IP address, not necessarily a specific user. Connections from remote sites using network address translation (NAT) [14], for example would allow multiple IP addresses access because NAT changes all internal IP address to a single external IP address. We needed to authenticate individual users rather than IP addresses.

The authentication methods used by the web firewall vendors we investigated [1, 2, 3, 4, 5] were deemed unacceptable. Their methods were either unacceptable from a security standpoint, or their authentication scheme did not fit into our existing authentication methods. Consideration was given to the HTTP CONNECT [12] method, which tunnels HTTPS and bypasses normal application layer functionality. This method presents a set of security issues, which we choose to avoid. Final design choices were two-factor authentication, a transparent proxy, and a secure authentication methodology. Implemented correctly, these would guarantee reliable access only to authenticated users and not IP addresses.

This authentication service, at a minimum, must provide the following capabilities. First, the proxy must be able to tell if an HTTPS request comes from an authenticated user session. Second, if the HTTPS request does not include valid authentication credentials, the proxy needs the URL of a web site that handles an initial, interactive user login.

There are two popular methods for keeping track of authenticated web sessions: one camp uses Kerberos tickets [16], (RFC 2478) [10], and one uses cookies [6, 7, 8], RFC 2965 [9]. Since the authentication system determines what is allowed through the proxy, careful design [17] and implementation is crucial.

LANL had an existing cookie based central web authentication service with the following security features:

- Two-factor CRYPTOCard [11] tokens with one-time passwords are used instead of traditional reusable passwords.
- The cookie sent to the browser is not persistent.
- The cookie string is random and contains no intrinsic value or information beyond being reasonably unique and hard to guess.
- The browser is instructed to send the cookie only to "*.lanl.gov" sites and only within an SSL session.
- The cookie is bound to the browser's IP address.
- Every new HTTPS request causes the cookie to be re-verified.

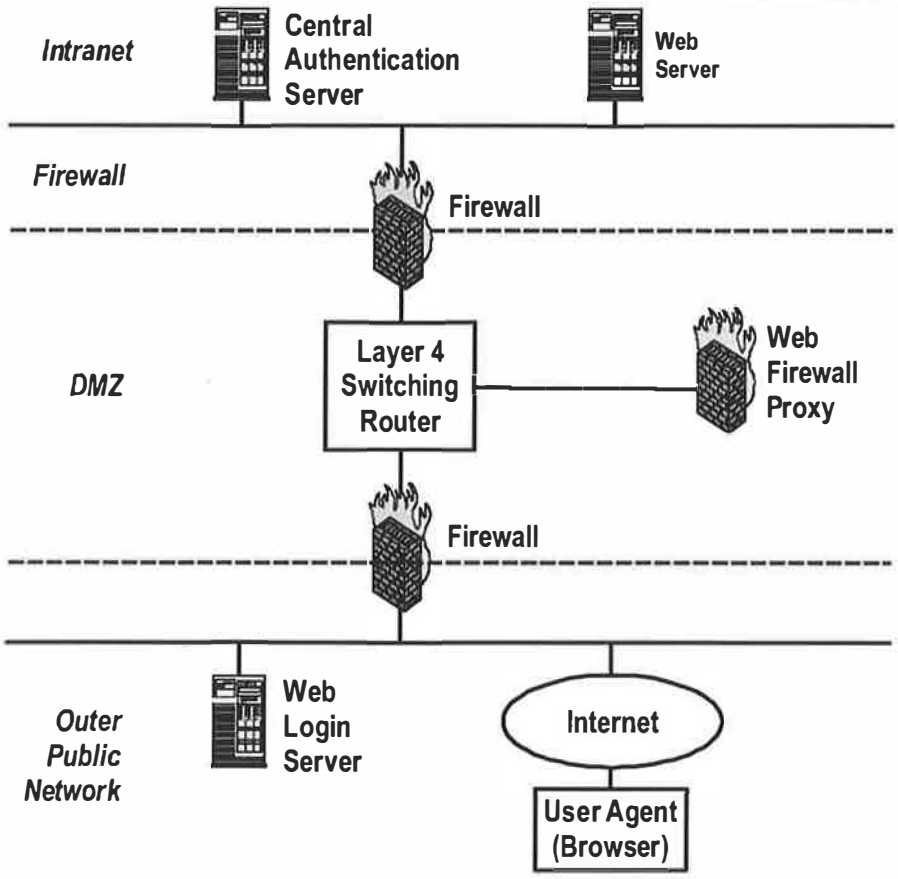


Figure 3: Web firewall architecture.

- To address the risk posed by a user walking away from a logged in session at an Internet kiosk, cookies have a maximum server-side lifetime after which the user must login anew.
- Cookies are also invalidated (server-side) if they are not used (idle time out).
- The user can voluntarily log out.

These features lent themselves well to the authentication needs of the proxy so this preexisting system was used.

Implementation

Our actual proxy implementation is comprised of several parts. See Figure 3, Web Firewall Architecture for details.

The first part requires the use of policy based routing to deliver HTTP/HTTPS messages to the proxy server. The firewall Demilitarized Zone Network (DMZ) [15] is contained within a commercial router that provides layer-4 policy routing. Incoming packets addressed to URLs behind the firewall are policy routed to the proxy server using layer-4 rules. All destination IP traffic addressed to any IP address behind the firewall going to ports 80 or 443 is policy routed to the web proxy.

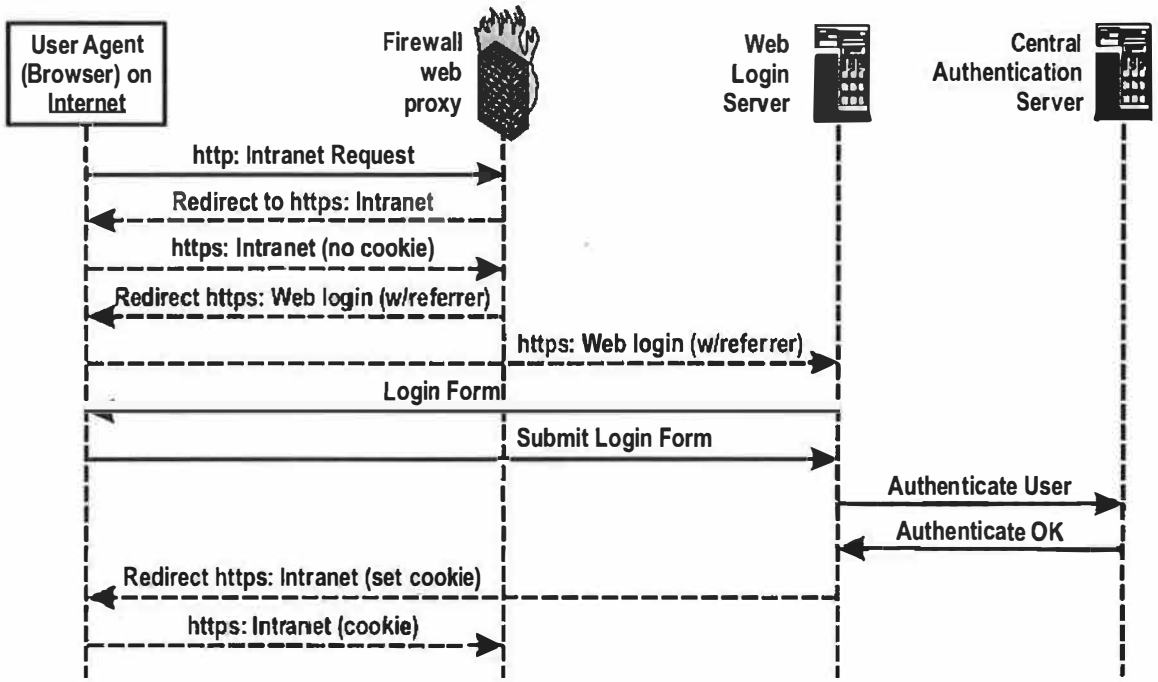
The second part is the web proxy server platform. We use the Linux operating system. Netfilter and iptables are part of the operating system. Netfilter runs within the kernel. Iptables provides the configuration interface to netfilter [14]. As described earlier, using iptables redirect, all packets policy routed to the web proxy are redirected to the web proxy. The proxy then masquerades as the internal web server when responding to the external web client.

The third part is the web authentication system. LANL's web authentication system is single sign-on. The session cookie obtained when the user authenticates works with all participating intranet servers. User authentication is two-factor authentication using CRYPTOCards [11]. This type of authentication makes use of one time passcodes. Following successful authentication, the password generated from the CRYPTOCard may not be used again. Our cookie attributes are:

```
Name = "SessionID",
VALUE = "Authentication random number",
Domain =".lanl.gov", Path ="/",
Max age = 0,
Secure = "1".
```

All other attributes are default values. The Max age attribute keeps the cookie from being stored on the web browser and causes the destruction of the cookie when the web client application is terminated. The secure attribute keeps the cookie from being transmitted in a non SSL request.

Cookies provide a security plus: the ability to revoke the authentication cookie at any time. Figure 4, Initial HTTP Request and Authentication, describes an initial connection without a valid authentication cookie. Figure 5, HTTP Request with Valid Authentication Cookie, describes a normal connection request that has a valid cookie. When an HTTPS request is forwarded though the proxy, the cookie is passed along too. Checking to see if a cookie is valid on the intranet



Note: Dashed lines are the result of redirects

Figure 4: Initial HTTP request.

servers can be tricky if the cookie is associated with the browser's IP address. The intranet web server sees the proxy as the user agent, but the cookie is associated with the end user's IP address. Our solution is to trust any cookie with a non-intranet address originating from the proxy. Integrating the proxy with other authentication systems may require a different approach.

- Our web authentication system uses two Linux servers: a login web server to provide the interface allowing the entry of a CRYPTOCard-generated password, and a back-end authentication server to validate the password.

The fourth and primary component of the system is the web proxy daemons. They handle redirection of HTTP to HTTPS requests and the actual connection proxying. The proxy consists of approximately 2500 lines of C code. The first daemon, called REDIRD, listening on port 80, redirects port 80 intranet requests to port 443. The second daemon, WFD, handles the port 443 connections. WFD handles header parsing to collect the authentication cookie and response codes, connections to intranet web servers, connections to the central authentication server to verify cookies, and bi-directional forwarding of allowed browser and intranet web server traffic.

Reliability

In general, issues that affect other Internet service availability will apply to the proxy. The usual hardware, operating system, server software, networking and power concerns are relevant and are not discussed here. Attacks from the Internet, however, present some different problems for the proxy.

The good news is the proxy does not need to be directly addressable from the Internet. Policy routing directs only port 80 and port 443 traffic destined for intranet hosts to the proxy. The proxy itself cannot be scanned or attacked in the usual ways.

The bad news is that the proxy must respond for all intranet web servers. When someone scans your intranet address space for ports 80 or 443, the packets are routed to the proxy. The impact to a single host from this type of scan is minimal. In the case of the web proxy, scans are multiplied by the number of intranet web server addresses which are policy routed the proxy. Under the right conditions, a simple scan turns into a denial of service attack on the web proxy.

For example, suppose the proxy is set up to respond for and entire class B network to avoid the administrative chore of tracking intranet web servers. A network scan of the site's intranet web servers will send 64K requests to the proxy. This generates a bit of extra work, but the proxy can keep up as long as the requests are sequential and use 3-way TCP handshakes. If the scan involves a burst of SYN packets sent to 64K addresses, it turns into a SYN flood attack. Connection queues for the proxy's daemon fill up so additional connection requests are dropped.

One solution is to throttle connection requests from individual source IP addresses. Since the proxy is responding for many destination addresses, the rule to limit connections applies to any destination IP address. A second alternative is to dynamically block source IP addresses sending malicious packets. Finally, to minimize the problem, one can limit the number of addresses the proxy responds for to real intranet servers that need to be accessed from the Internet. This also improves security.

Performance

The proxy software makes light demands on the hardware. The application does some network I/O, some string manipulation, and SSL encryption. A modestly configured PC should be able to keep up with most site's workloads. Our proxy runs on a 800 MHz Intel Pentium with 512 Mbytes of memory and 100 Mb/s Ethernet and enough disk storage for a minimal Linux operating system and logging. With this hardware and current workloads, there is no user detectable difference in response in fetching web pages through the proxy and from within the intranet.

The web proxy currently proxies for one class B address range. There are approximately 1000 individual users who access the web proxy. The proxy typically handles 10,000 to 12,000 authenticated requests per day.

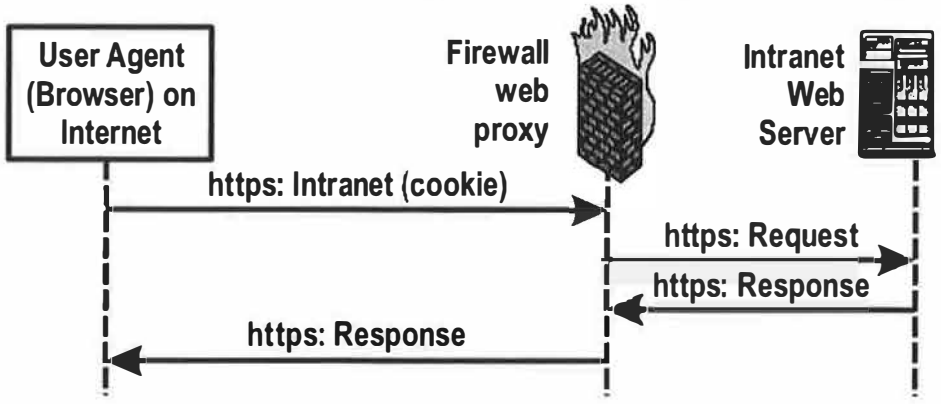


Figure 5: HTTP request with valid authentication cookie.

The average number of simultaneous connections is 12 with peaks to 36. We have seen no performance issues since installing the proxy. Port 80 and 443 scans of the class B address space do on occasion cause a slowdown in the proxy's response to valid connections.

The best way to improve performance is the same as improving availability: keep the Internet noise away from the proxy. Also, a layer-4 switch or policy router using proxy load balancing would allow for multiple proxies, resulting in scalable performance and improved availability for large applications.

Topology

The web proxy service consists of three pieces: a web login server, an authentication server (e.g., a Kerberos key distribution center-KDC), and the proxy itself. Where you place these components in your firewall architecture involves some security trade offs. The weblogin server, a real web server, is the only component that needs to be addressed directly from the internet.

Our weblogin server is outside the firewall, where it is subject to attack. If compromised, the intruder can capture names and passwords of users on the weblogin server and use them to get access through the web proxy. Intruder access is limited to firewall services and some users' accounts.

Placing the weblogin server inside the firewall with a special rule permitting access from the Internet creates a bigger problem if the server is compromised. The intruder is no longer limited to firewall services for certain users; you have an intruder inside your intranet. Our authentication server is inside the firewall, but is not directly accessible from the Internet. With the weblogin server outside the firewall and the authentication server inside, we have a special rule (hole) that allows them to talk. Capturing the weblogin server allows the intruder to attack the authentication server.

Issues

Forcing the use of SSL was recognized to present issues to user agents that the end user must deal with. Some user agents will complain about wildcard certificates and the user must specifically accept the certificate. We have observed some vendor proprietary systems using HTTP protocols that have been coded never to accept wildcard certificates. Another issue arises when system administrators have image source tags that are absolute, rather than relative. When the image tag for example is http://www.foobar.net.com/image, the image will not always display properly. Because we have forced the user agent to use SSL, it should comment that it is about to fetch an insecure document, and give the user the option to proceed or not. Instead, some agents will silently ignore the image tag, resulting in an incomplete page being displayed to the external web client.

Web servers that run both secure (HTTPS port 443) and insecure (HTTP port 80) on the same server will cause problems. When the user connects to the server on port 80, and the proxy has established the session with HTTP keepalive active, and then the user selects a link on the returned page sending them to the HTTPS server on the same web server, the proxy will continue to proxy the request to port 80. It will receive an HTTP 404 response code and can recover from this error for any request types except posts. In these cases the proxy saves the original request and retransmits the saved request to port 443. However, due to the format nature and potential large size of post requests, the original post is not kept and the request will fail. It is not conceivable to save and reissue the post requests. In practice, however, this has not actually been a problem for us because our web server system administrators do not operate their servers in this manner.

As written, the proxy only supports standard HTTP ports 80 and 443. Other ports could be supported, but would require additional code. One way to address this would be to maintain a configuration file with a list of servers and associated non-standard ports.

For those who are so inclined, one can use self-signed certificates, as opposed to paying for certificates signed by recognized organizations. Doing so will generate a warning message on the browser, requiring the user to purposely accept the self-signed certificate.

Revocable authentication credentials is a valuable security tool. We limit the authentication credential lifetime. When the cookie expires, whoever is sitting at the browser has to login again. Inactive sessions are closed. If the proxy does not hear from the browser within the idle time out interval, the user must login again. Finally, the mindful user can explicitly logout or simply exit the browser; both remove the authentication cookie from the web client.

Even with all the built in security protections, it is still possible a session could be compromised. Good logs are critical for detection and recovery. The proxy logs all successful transactions with the source and destination IP addresses, the user's ID, and the HTTP request. The logs can be fed to a real time anomaly detection system and reviewed manually. Previous login successes, failures and logouts can be displayed after a successful weblogin so the user can report unexplained activity.

Security Analysis

Given the trusted nature of the web proxy, potential security risks are of particular concern. Below is an attempt to identify and show mitigation efforts for such concerns.

First, we must deal with network-based risks. Given that the web proxy is an inbound firewall service proxy, the following three network-based risks are salient [15]:

- Session hijacking
- Packet sniffing
- False authentication

Session hijacking is best mitigated by the use of SSL/TLS connections for all data movement beyond the initial referral from port 80. Since the proxy presents a properly (VeriSign) signed wildcard certificate, users can be relatively assured that a man-in-the-middle situation is not ongoing. As indicated in the SANS SSL Man-in-the-Middle Attacks paper [19], using older Internet Explorer browsers can make it easier to accomplish man-in-the-middle attacks.

Packet sniffing is also well protected against by the encryption provided by the SSL/TLS connections. Usernames, passwords, and subsequent authentication cookies are all contained within the encrypted channel. The proxy does continue to support shorter key lengths for older international browsers, mostly from a capability need. Currently, we feel the capability need outweighs of the risks of the reduced encryption quality, though such decision may be reversed in the near future.

Finally, false authentication risks are well mitigated by several aspects of the system. The use of the short-lived authentication cookies, the relationship of the cookie to originating IP address, and the random content of the cookie mitigate the risk of stolen or forged authentication cookies. The use of two-factor generated one-time passwords significantly lowers the risk of stolen passwords.

The transparent nature of the proxy may give some level of protection through its obscurity and difficulty in understanding the proxy's relationship to the web servers it sits in front. While the obscurity benefit may be minimal, it can definitely make scan probes of the internal network look unusual. One negative concern related to this transparent aspect is potential denial of service to the proxy, both intentional and unintentional. Since the server answers for a wide range of addresses (class B in our case) on port 80 and 443, there is significant potential for overwhelming the server. Experience has shown this can be mitigated through the use of SYN cookies [20] and keeping the port 80 redirector lean and simple since it receives the majority of denial attacks. Active methods of intrusion detection and automatic host blocking have also mitigated the impact of scans.

A second concern is logic and programming errors. These are the standard inherent low-level system risks relevant to any security-centric application. These risks include logic errors (e.g., unintended granting of authentication cookies) and programming errors (e.g., buffer overruns). The resulting unauthorized access could be either unintended access to the protected websites or administrative access to proxy itself. Obviously such risk must be addressed.

A multi-tiered mitigation approach to such errors has been taken. First, the proxy has undergone both peer design and code reviews to help remove both logic and programming errors. Formalized system configuration (we use a combination cfengine/cvs approach) including a minimized system install, helps to ensure a potential compromise has few other applications to leverage. Helping to ensure least privileged access for the proxy to the internal network, the router ACLs only allow the proxy to have TCP ports 80 and 443 access, again mitigating exposure should the proxy application or server be compromised. Minimizing vulnerabilities on internal web servers also reduces the risks of a compromised proxy. This mitigation is done with regular vulnerability scans of the internal network web servers. One should check the code using tools like Flawfinder and Rough Auditing Tool for Security (RATS). These tools will possibly identify security problems with the code. The proxy also performs off host system logging. To date, there have been no known compromises of the web proxy system.

The last aspect of security concern involves end-user behavior. User behavior can make or break any good security stance. A major security concern with the web proxy is user error. The proxy allows authenticated users access intranet services. The service is designed for users coming from public systems they do not control like kiosks, cyber cafes, or conference terminal rooms. Many users do not understand web cookie management or are forgetful. Walking away from a "logged in" browser is a real possibility and concern. The ability to revoke credentials is an important feature, as is their max age attribute of zero.

Conclusion

Users have received the proxy positively; with the LANL proxy handling over 10,000 authenticated HTTPS requests per day. The fact that the proxy is transparent, not requiring any browser preferences to be set, is particularly beneficial. This feature allows users to access LANL intranet web sites from anywhere they have access to a web browser. In addition, the per-user authentication has worked well. We have encountered no technical problems with the use of authentication cookies and no security issues with this mechanism have been detected. The proxy's simplicity and non-intrusive nature into the end data exchange has allowed it to work effectively with Java, Javascript, and all other web-based application extensions currently used and foreseeable. Overall, the proxy provides a highly flexible yet secure mechanism for Internet users to access intranet web content.

Author Information

James (Jim) Clifford is the Network Service Team Leader and a Systems Software Engineer for the Network Engineering Group at Los Alamos National Laboratory. His interests include Internet technology, Linux, and practical computer security. Jim has a BS from the University of Michigan. He may be reached

view e-mail: jrc@lanl.gov or smail mail: MS B255, Los Alamos National Laboratory, Los Alamos, NM 87545.

Alexander (Alex) Kent is the Deputy Group Leader for the Network Engineering Group at Los Alamos National Laboratory. His primary development projects include Laboratory-wide authentication and user account systems, network information propagation and architecture, and the Los Alamos firewall system. Alex has an BS and MS in CS from New Mexico Tech and an MBA from the University of New Mexico. He may be reached at <alex@lanl.gov>.

Roger Crandell is a Staff Member in the Network Engineering Group at Los Alamos National Laboratory. His primary role is firewall development and software engineering. Roger holds a BSEE from the University of Nebraska. He may be reached via email at rwc@lanl.gov.

References

[1] *Checkpoint FireWall-1*, http://www.checkpoint.com.

[2] *Cisco PIX firewall series*, http://www.cisco.com.

[3] *Secure Computing Sidewinder G2*, http//www.securecomputing.com.

[4] *Squid Web Proxy Cache*, http://www.squid-cache. org.

[5] *Netscreen 5000 series,* http://www.netscreen.com.

[6] Web Initial Sign-on (WebISO), http://www.middleware.internet2.edu/webiso.

[7] Thomas, Stephen, HTTP Essentials, 2001.

[8] Krishnamurthy, Balachander and Jennifer Rexford, *Web Protocols and Practice*, 2001.

[9] Kristol, David and Lou Montulli, "HTTP State Management Mechanisms," *RFC 2965*, IETF, October, 2000.

[10] Baize, E. and D. Pinkas, "The Simple and Protected GSS-API Negotiation Mechanism," *RFC 2478*, December, 1998.

[11] CRYPTOCard Corporation, Kanata, Ontario, Canada, http://www.CRYPTOCard.com.

[12] "Expired IETF Internet-Draft," Art Luotonen, 1998.

[13] "ITU-T Recommendation X.509(03/00): Information Technology-Open Systems Interconnection – The Directory: Public-key and attribute certificate frameworks."

[14] Ziegler, Robert L., *Linux Firewalls*, 2001.

[15] Chapman, Brent D., and Elizabeth D. Zwicky, *Building Internet Firewalls*, 1995.

[16] http://www.globecom.net/ietf/draft/draft-brezak-spnego-http-03.html.

[17] http://www.pdos.lcs.mit.edu/papers/webauth%3Asec10.pdf.

[18] Marsh, Matthew G., *Policy Routing*, http://www.policyrouting.org/PolicyRoutingBook/ONLINE/TOC.html.

[19] http://www.sans.org/rr/papers/60/480.pdf.

[20] Lemon, Jonathan, "Resisting SYN flood DoS attacks with a SYN cache," *Usenix BSDCon*, 2002.

Designing, Developing, and Implementing a Document Repository

Joshua Simon – Consultant
Liza Weissler – METI

ABSTRACT

Our company had grown large enough and complex enough to require a centralized repository for its documentation. Various internal groups produced all sorts of documentation; regions and districts produced documentation for and about clients, including proposals for work and the results of work performed. Often the documentation produced did not take advantage of previously-produced work.

The Repository project intended to centralize all documentation in a single "one-stop shop" for creating, updating, storing, and searching documents, to provide various information about every document so documentation authors and information gatherers could search for and use or reuse existing similar documentation as appropriate. It also was intended to be minimally intrusive for both the authors and the users of the documentation. This allowed the company to enforce a common look-and-feel for all documents within a certain type and for consistency in content as well. Other goals included the ability to update content in exactly one location and have that change propagated throughout all relevant documentation.

The user interface had to be simple enough for nontechnical people (managers, sales staff, administrative assistants, and so on) to be able to use it to add, edit, and search for documents meeting their variable criteria, and yet complex enough to be meaningful in terms of the results. Because of the high costs of solving this problem correctly, we decided to take a low-key internal approach to the project instead.

Introduction

Problem

While working as consultants, our former company had grown large enough and complex enough to require a centralized repository for its documentation. Various internal groups, such as Finance, Information Technology, Legal, Marketing, Recruiting, and Sales, produced documentation. Regions and districts produced documentation for and about clients, including proposals for work and the results of work performed. Often the documentation produced did not take advantage of previously-produced work. The Repository project intended to centralize all documentation in a single "one-stop shop" and provide various information about every document so documentation authors and information gatherers could search for and use or reuse existing similar documentation as appropriate.

There's an old adage about the shoemaker's children never having good shoes. Similarly, even though we were a company of system administrators, we often needed to design our own tools. Most everyone in the company recognized the need for such a repository but nobody had the cycles to design or implement one. Josh was moved into a corporate-side role to work on this project, given his extensive history of documentation and project management work, as well as the demonstrated ability to translate between technical and nontechnical staff. Josh asked Liza for her assistance on the database and Perl side, since she was a member of the tool development team and already familiar with the internal IT department policies and because she has a background in library sciences.

Because of the nature of the business and her own responsibilities, Liza could only work on the project during her off-hours. Similarly, we were constrained from spending any real money on hardware or software; even though most technical people and many nontechnical people wanted a repository, nothing was budgeted to pay for it.

Another constraint was the term *documentation repository* having different meanings for different people. The term could mean an electronic library like the physical one at Alexandria, or a vault containing copies of the physical documentation we'd produced. But defining *documentation* was itself a hard problem. Certainly any physical document we'd deliver to a customer counted – such as summaries, analyses, recommendations, project plans, and so on. Similarly, the pre-project work, such as statements of work, service level agreements, and other contracts count as documentation. But what about the online question-and-answer database? What about marketing materials and sales literature? What about the HR-specific forms (such as the IRS I-9 and W-4, and the annual reviews of your employees and management)? As we looked

further into the back office within the company we found that more and more document types could and did exist. Our repository had to allow for those document types. This led to our categorizing system.

While we could have just added disk space onto a centrally-located file server, we had many different people creating and editing documentation, many of whom used different operating systems and different tools for producing or updating the documentation. Furthermore, different people use different naming conventions and different organization or classification schemes, so finding something without a search engine of some kind would be exceedingly difficult.

Goals

The major goals of the project were to provide a single place on the company network for all documentation to be created, updated, stored, and searched, while being minimally intrusive for both the authors and the users of the documentation. This should have allowed the company to enforce a common look-and-feel for all documents within a certain type and for consistency in content as well. Minor goals included the ability to update content in exactly one location and have that change propagated throughout all relevant documentation.

Requirements

We spent a substantial amount of time identifying the requirements for such a documentation management package. These included:

- the ability for anyone to add documents to the repository
- the ability for document owners to edit the documents
- access control for who could read or make changes to documents; for example, only the specific employee, management, and Human Resources should have access to financial information
- strict restrictions on deleting documents from the repository; in general, we were going to forbid the delete function to all except specific "librarians" whose role included database content management
- keyword-based searching
- storing meta-data and a URL instead of the entire document
- no constraints on how people create, edit, or maintain the documents themselves; revision control is a solved problem we didn't want to deal with

We then began researching the vendors in the space and the costs involved. Because of the high costs of solving this problem correctly, we decided to take a low-key internal approach to the project instead.

The interface itself had to be simple enough for nontechnical people (managers, sales staff, administrative assistants, and so on) to be able to use it to add, edit, and search for documents meeting their variable criteria, and yet complex enough to be meaningful in terms of the results.

Being system and database administrators really made this project possible. As system administrators we'd had previous experience with talking intelligently to vendors' sales personnel. We're skilled at converting what business (usually nontechnical) people want into and out of technical terminology. It's typical, in our experience, for system administrators to fill programming needs when there are no other "official" programmer resources available for urgent project work. Having a database-savvy member of the team saved us a lot of unnecessary work as well.

Design & Development

Based on the requirements analysis and the reduced necessary functionality, we decided to track the following information:

Author: The authors of the document; we found that being able to search for documents by author was useful since people would remember "Alan or Betty wrote ..."

Title: The document title

Description: A short description about the document, which would show up on search results

Revision: The revision number or string, such as "1.0" or "First Edition," to allow for tracking multiple copies of a document over time

Date: The date the document was published or released (which could theoretically be in the future)

URL: to the document, including the protocol specification

Permissions: The permissions and state information: whether the document has been approved for release or was a draft, whether changes can be made to the document, whether the author has checked it out for revisions, and whether public (meaning unauthenticated) users could access the document

Search flags: Company-internal codes for what major and "Networking" and "Security"

Class: The document class, one of client/technical, internal documentation, marketing/sales literature, recruiting, or other; these five categories were a direct result of our analysis of defining a document

Subclass: The document subclass, a well-understood company-internal code to further classify the document, different for each class

Keywords: A list of keywords to be used in searches; for cross-application consistency we used the same keywords for the repository as we did for the question-and-answer database

A sample record, using this paper for content, would be the following:

Author: Simon, Joshua; Weissler, Liza
Title: Designing, Developing, and Implementing a Documentation Repository
Description: A LISA 2003 paper on how we designed, developed, and implemented a general-purpose documentation repository
Revision: 1.0
Date: October 26, 2003
URL: http://www.usenix.org/publications/library/proceedings/lisa03/tech/simon.html
Permissions: Approved, Public [structure of bits: approved 1, can-change 0, checkedout 0, public 1]
Search flags: null
Class: I (internal)
Subclass: D (documentation)
Keywords: doc, repo

We had certain design elements determined from a recent documentation styles analysis project that defined fonts, relative sizes of headers and text, use of color, and so on, which made some elements of the user interface design fairly straight-forward.

The top-level or home page contains a list of the nine major topics:

- **Document-specific:**
  - Adding a new document record to the Repository
  - Editing an existing document record in the Repository
  - Uploading a new document to the Repository
- **Searching:**
  - Browse the entire Repository for document records
  - Searching for a document
  - Jump to the Templates page for the latest document templates
- **Repository documentation:**
  - User's Guide and Administrator's Guide
  - On-line Help
  - Get statistics on documents in the Repository

Each subsidiary page would contain navigation links in the header (including a link back to the Repository home page, a link back to the intranet home page, and links to related functionality in other applications on the web), the page content (such as data entry form, results, or error message), the application revision number, and the standard company footer. The standard footer provided contact email, a link to the company's internal bug reporting application, and the copyright statement.

Development of the application took place over a two-month period (February 7 through April 5). The application was written in Perl (5.005) using the DBI and DBD::Oracle modules, with Oracle 8.1.6 for the back-end database. Development and quality assurance testing were on systems running Red Hat Linux 6.2, while the production system was running Solaris 2.6.

Staffing of the development effort was basically one person in her spare time, so it may be said that the development budget was essentially zero (which yields quite a good return on investment). Our company had a long history of infrastructure building as volunteer effort on the part of employees; this helped the company keep costs down, and gave employees the sense of having more of a stake in the company. Of course, it also had some influence on the distribution of yearly bonuses. More critical applications were the "day job" of a team of virtual developers, but as this application fell a little lower on the priority list, it remained a volunteer effort.

We don't have hard and fast data on the number of person-hours spent on this project. As noted in the "Acknowledgements" section, 58 members of the company worked on this project either directly (design, development, and testing) or indirectly (related development, development of modules we could adapt, etc.). Josh spent probably four months working on this project, including the detailed requirements analysis, and Liza spent about one month on it.

The application itself was fairly quick to code due to a number of existing Perl modules in use at the company to standardize database access, authentication, error reporting, and so forth. The programmer herself was one of the company's "virtual developers" and thus was ramped up on the company's application design philosophy and structure. (Had this effort been a real job and not a volunteer effort, it likely could have been developed in well under half the time.)

The documentation that goes along with the code – both a User's Guide for the general user (extensive online help) and an Administrator's Guide for the librarians – was written alongside the application. Having a standardized cascading stylesheet for both the application and its documentation helped maintain the common look-and-feel across both. The documentation development time is included in the 6-months of application design and development time.

Concurrent with the application and database development, we collected bibliographic data on existing company-wide web-based documentation so we could force-load the Repository with valid data. This was an interesting process in itself, as the data could be almost anywhere. A simple conversion script parsed the bibliographic data, performing error checking and loading the data into the database.

Our system administration skills were essential to the success of this project. We analyzed the problem, which (as with many projects in technology)

seemed to get bigger and more complicated the more we looked at it. We identified our requirements so we could determine what we wanted as well as what we didn't want. We used organizational skills to figure out how best to allocate our volunteer resources to best effect. We documented the application, both in the code and in standalone User's and Administrator's Guides. Our familiarity with database schema design, library sciences, scripting, and best practices in software development and testing as part of the release cycle was absolutely essential in producing a usable, functional application.

Testing

The testing process took place in three major steps: alpha testing, quality assurance testing, and beta testing. Our rationale for this was four-fold. First, we wanted the people who would become the content consistency police, or the librarians as they were called, to be intimately familiar with the processes involved. Second, we wanted to make sure, before releasing the application to the whole company (including both technical and nontechnical staff), that both technical and nontechnical people could use it. Third, the IT department had a dedicated test group (off-hours volunteers) who would do our QA testing for us. And fourth and finally, it's good software development practice to do three-stage testing (alpha, QA, and beta).

The company had three full-time technical writing staff. We used them as our alpha test team, gave them full access to the database, and asked them to test everything: adding, editing, and deleting documents. Since they were using a test database (which would be purged and reloaded with real data before production), we let them know deleting records was okay for the alpha test, though we asked them not to do so with "real" documents once we'd gone live. During the two-week alpha process we answered four email questions and reported a total of 26 bugs or feature requests, most of which were fixed before the QA test period began.

Immediately after the alpha period we went to a Quality Assurance (QA) period. QA was handled internally by a team of volunteers called "Bugzappers." The process required members of the team to say whether they'll participate in a release, and if so, to review the application specifications and documentation, test features, report bugs, make feature requests, confirm whether bugs and feature requests purported to have been handled actually were, and finally, give a yea or nay as to whether the application can be released. We had the participation of about half a dozen Bugzappers who picked nits with the application during the QA period. During the QA period our test team identified 13 new bugs, all of which were fixed before beta. At the end of the QA period, only six bugs or feature requests remained open.

To perform a wider test we released a beta version of the application and database to the documentation-focused experts within the company. This included the alpha and QA test teams, as well as the Publications committee, who oversaw and mentored all technical writing within the company; the Methodologies Development team, who designed, developed, and documented detailed methodologies; and the Repository group, who were responsible for any and all thoughts on documentation and tool repositories company-wide. During the week-long beta test the team identified 21 new bugs or feature requests, most of which were fixed before production. At the end of the beta test period, only 17 bugs and feature requests remained open.

Table 1 shows the bugs, feature enhancement requests, and user training issues during the testing process.

Type	During Dev & QA	During Beta	Entire Process
Bugs	30	6	36
Enhancement requests	7	7	14
User training issues	6	3	9
Total	**43**	**16**	**59**

Table 1: Bug status.

Production Implementation

Once the beta testing was complete, we put the application into production. Doing so was as simple as announcing to the company as a whole that the application existed and putting a link to it on the main intranet home page.

Within the first month in production, users identified some 15 additional bugs and feature enhancement requests, all of which were later implemented, such as tying the documentation more closely to the application; cleaning up the user interface; and providing automated email notifications to the librarians when records were added, edited, or deleted to ensure sanity in the database.

A month after the initial release (called 2.0 for historical reasons) we released a patched version (2.1) that fixed four of these issues. Additional patch and incremental hot-fix releases over the next four months fixed or implemented an additional four of 13 bugs or requests. Table 2 shows the time line of the testing and release cycles.

Measuring Success

It is difficult to measure the success of this project in a quantitative manner. Other than the number of documents, the number of users, and the

number of searches, there is no real quantitative measurement of success. These measurements are discussed, along with qualitative measurements, in the following sections.

Version	Date	Description
x2.0.0.1	Apr 5	Alpha test
x2.0.0.2	Apr 13	Quality Assurance
x2.0.0.3	Apr 23	Beta test
x2.0.0.4	Apr 27	Beta test, continued
v2.0	May 7	Initial production release
v2.1a	Jun 9	Rearranged top-level page, added Upload and Templates links, added version to footer
v2.1b	Sep 7	Corrected typos and formatting
v2.1c	Oct 4	Cosmetic style changes for application & documentation
v2.1d	Oct 15	Bug fix
v2.1e	Oct 19	Bug fix

Table 2: Incremental release time line.

Number of documents

The number of documents – or in reality, the number of unique document URLs – in the database grew over time. The initial bibliography collection phase, in fact, allowed us to begin production with nearly 800 documents. Table 3 and Figure 1 show the number of documents over time.

The continued growth of documents in the Repository after going into production, not all of which were added by the librarians as an extension of the bibliography collection process, showed us that the Repository was indeed useful. This met our expectations.

Date	Event	Documents
Feb 24	Original coding	3
Feb 28	Bibliography collection phase 1	346
Mar 15	Bibliography collection phase 2	567
Apr 15	Bibliography collection phase 3	752
May 7	Initial release	794
Jul 17	Revisions announced	897
Oct 23	One author left the company	1101
Dec 4	Other author left the company	1128
Dec 18	Insider source reported statistics	1130

Table 3: Documents over time.

Number of Users

The number of users also grew over time. This met our expectations, as we continually grew the number of possible users, as shown in Table 4.

Phase	Users
Development	2
Alpha test	5
Quality Assurance	11
Beta test	34
Production	> 250

Table 4: Number of users over time.

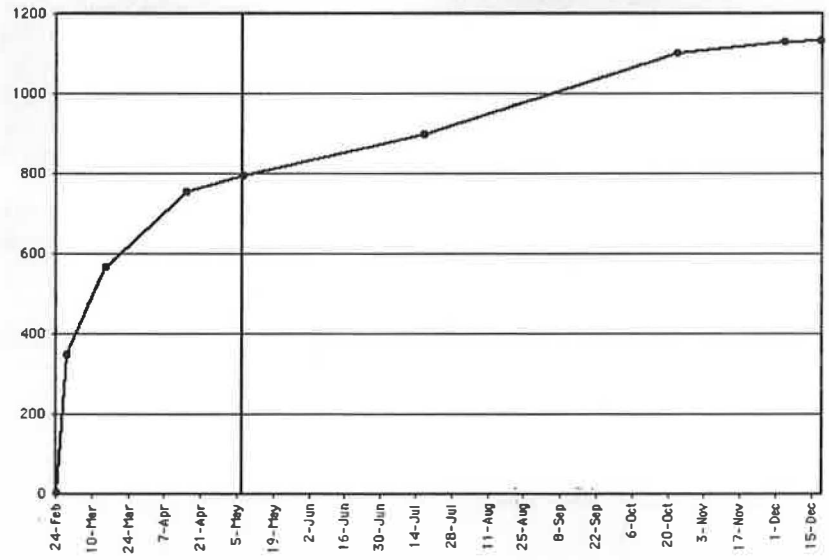


Figure 1: Documents in respository over time.

Between the start of the beta and the start of production, the company experienced several rounds of layoffs, reducing the maximum possible audience from over 400 to around 250. That maximum number continued to decrease during the production period down towards 150-200. This curtailed the growth pattern in usage and also reduced our ability to research and analyze the data further.

Number of Searches

We do not have data on the number of searches performed, as the httpd access_log files have not been made available. Both of the authors of this paper left the company through layoffs, and the company has not produced the old logs for this paper.

However, even were this information be made available it would be difficult if not impossible to draw any meaningful conclusions from it. Searches during the development and testing phases are meaningless, since they are intended to test both successful and unsuccessful searches. Searches made during production might be meaningful if we tracked success (defined as the search resulting in either at least one document or the document(s) the user wanted or needed), but we did not do so early enough to collect the data. Finally, even if we knew how many searches were performed (regardless of success), the large reduction in the user base makes most of the numbers meaningless.

Qualitative Measurements

It is also difficult to measure success qualitatively. Without access to the log files, and without a survey as to whether the users were satisfied with their use of the Repository, there is no real qualitative measurement that was performed. Feedback was generally positive, with bugs and enhancement requests submitted by several users; several members of the company, in various roles (such as technical, sales, marketing, and executive management), mailed to thank the development team for their efforts and praising the Repository.

Related Work

There were several intranet document management systems when we first started looking at vendors (over two years ago as of the date of first publication of this paper). Based in part on a document management seminar one of the technical writers attended, there were 37 software packages on our "long" list, including XML development packages. We realized before we could narrow down to a "short" list that we could not afford any of the packages, though the most likely candidates included Documentum, Enigma's DynaText, IntraNet Solutions' Intra.doc!, QUIQ's Knowledge Network, and Vignette's content management system.

We also briefly considered using wikis for this project. A wiki is a composition system-meets-discussion medium tool used for collaboration. While wikis are great for groupware or collaborative work on documentation development, we weren't developing documentation in this project. We were developing a repository to store and track documentation explicitly developed elsewhere. A wiki was the wrong tool for what we wanted.

Future Work

Future work includes fixing bugs identified during the continuing use of the Repository software as well as adding the additional features omitted from the current release. In no particular order, these include:

- Separating the one big index.cgi script into smaller, more-manageable chunks, say to have a search.cgi instead of index.cgi?action=search. The original intent was to have much of the guts of what the various actions do are collected into subroutines in a separate application Perl module, keeping index.cgi relatively lean and mean, but as the application grows it makes sense to break it apart.
- Re-think the company and non-company author handling, or at least automate record updates when an employee leaves the company.
- Provide a record-level authorization permission structure.
- Provide and track type-dependent information (e.g., publisher for books and magazines, magazine title and issue/volume/number for articles and columns, etc.)
- Include command-line interfaces to create new documents & templates, update existing documents & templates, search for documents & templates, administer the database, and convert a document from one format to another.
- Provide a file system-like interface for browsing.
- Provide tools to convert between a consistent back-end format (such as XML) and the desired front-end formats (HTML, PDF, RTF, TXT, etc.); for example, xml2html and xml2pdf and pdf2xml and html2xml.
- Provide a tool to check out (and lock) a document for revision.
- Provide an interface to the public Internet web site (so non-authenticated personnel can search for PUBLIC documents).
- Provide an interface for people to suggest changes to documents they do not own.
- Provide and track document-level security so users cannot see document records (including the URLs) for documents the web may not let them access.
- Provide a way to "group" documents.
- Provide a way to "group" permissions.
- Provide a better mechanism for documents to include or supersede each other. Updating data in one place should update it everywhere.

- Update the document record automatically if the content of a certain URL changes.
- Generate and track document numbers.
- Provide a means for a document to be broken off from a reference so changes to that reference no longer are applied to the document.
- Provide a custom query interface (build-your-own SQL).
- Provide a way to save and load complex queries.
- Enforce documentation process flow (creation, modification, deletion, permissions, change control, state changes, and so on).

Conclusions

Our major goal of providing a single all-inclusive starting point for searches and a repository for uploading documents was met. It was minimally-intrusive for both authors and users. Our minor goal of updating content in one place and propagating that change to other documents was not implemented in this version of the software but remains a possible enhancement.

What did we learn during this project? We learned how important ease-of-use and a clean user interface are for applications, both for document authors (adding and editing records, uploading documents) and searchers. We learned that different people map information in different ways, and that any knowledge mapping or searching system has to be usable regardless of how we do so.

We learned how important our system administration skills were. Requirement analysis, organizational and project management skills, discussions with vendors, being able to translate between business and technical people, and best practices in programming, software development, and testing were all essential to successful deployment of the repository. Documenting the application for both technical and nontechnical users as well as for the administrators helped the application gain momentum within the company. Having a database-savvy member of the team saved us a lot of unnecessary work as well. We learned that having a project plan, with specific milestones and goals, is essential. And we learned that preventing project creep can be done, provided that the project manager is firm and the requirements are well-documented and well-understood.

We would strongly suggest to anyone interested in implementing a repository that you identify what you want and don't want well in advance. Do not allow yourself the luxury of adding "just one more cool feature" unless it truly is something you need. Consider both free and for-pay software that exists before writing your own if at all possible. If you're in an environment which already uses some tracking or coordinating system (like the Class and Subclass we used here), implement it as part of your database. Talk to your users early and often; if you want them to use the tool they need to understand what's in it for them, what benefits it can provide, and how easy it is to use.

Availability

The source code for the Repository script can be made available upon request. The version deployed in our former company uses many internal code stubs which we are reworking to be generic. At present it still makes several assumptions, including that there is an Oracle database back-end and the appropriate CPAN Perl modules are available. We want to make the script a bit more robust and intelligent to allow for different databases and authorization schemes before publishing the software under some form of public license.

Acknowledgements

The authors would like to thank the following people:

- Mike Stok for developing the original upload script (version 1.0 of the Repository).
- Katherine Ross, Jeff Schouten, Jordan Schwartz, and Emily Stemmerich for the initial bibliographic data collection that let us preload the Repository with nearly 800 data records.
- Bill Huff and Tim Peoples for developing the underlying Perl modules that normalized database authentication and access.
- Cliff Nadler and Ryan Skadberg for their assistance, advice, and implementation ideas.
- Lee Amatangelo, Todd Chapman, Brian Clark, Katrinka Dall, Mark Dawson, Doug Freyburger, Marc Furon, Charles Gagnon, Jason Heiss, Barbara Howard, Gerard Hynes, Brian Kirouac, Ron O'Neill, Joe Royer, Tapan Trivedi, and Jeff Tyler for confirming the integrity of the data before we rolled the application into production.
- Angela Gatto, Brian Worrall, and Julie Zacharias for their tremendous assistance in drafting the initial requirements specification and for their alpha-testing the Repository.
- Alvin Gunkel, Barbara Ingram, Peter Pak, Keith Patterson, Tom Whitley, and Michael Wilson for their assistance in the quality assurance process.
- Ed Bailey, Chris Barnash, Bryon Beilman, Dave Bianchi, Matt Coffey, Steve Cruz, Ralph Dahm, Randy Dees, Ryan DiDomizio, Jim Flanagan, Jeff Giuliano, Mark Jones, David Leonard, Jim Niemira, Jason Powell, Michael Rice, Rodney Rutherford, Joel Sadler, Andy Silva, Ed Taylor, Rob Worman, and David Young for their extensive beta-testing of the Repository.

Furthermore, thanks to the dozens of people within the company who wrote documentation, uploaded it into the Repository, ensured the accuracy of the Repository records, and collected bibliographic

details of others' works to ensure completeness and accuracy of the data in the Repository. Your help brought us to over 1,100 valid document records in under six months of production use.

Finally, thanks to Alva Couch, Æleen Frisch, Tom Limoncelli, and Adam Moskowitz for their assistance in reviewing this paper. Your comments helped us tremendously, even when we didn't agree with them.

Biographies

Josh Simon has 12 years of experience in UNIX system administration, project management, and technical writing. He has a long history of contributions to SAGE, including serving on the SAGE Executive Committee, being the Desk Editor of SAGEwire, chairing the SAGE Online Services Committee, and serving on five LISA program committees. He's written and coordinated summary writeups for several USENIX Annual Technical Conferences and LISA conferences in the past for publication in *;login:*. His non-technical interests include cooking, reading mysteries and science fiction, and plotting to take over the world. Reach him electronically at jss@clock.org.

Liza Weissler has 17 years of experience in UNIX system administration, Oracle database administration, and application development. She worked for the RAND Corporation (Santa Monica, CA) and Collective Technologies before giving up her Southern California native status and moving to southeastern Arizona. She is now happily ensconced in the foothills of the Huachuca mountains, is employed by Management and Engineering Technologies International Inc (METI; www.meticorp. com), and works as a contractor to the US Army's Network Enterprise Technology Command (NETCOM) at Fort Huachuca, Arizona in between vacations. Contact her electronically at liza.weissler@us.army.mil.

Appendix A: Database Schema

Overview

This is the sql used to create the "repo" database in Oracle. Basically the tables are:

- **documents** – The main table
- **states** – Where the public/blessed/locked/frozen bits are defined
- **codes** – Internal codes
- **class, class_type** – Define class/subclass
- **authors** – Table to store info on folks no longer with the company

Of the others: long_data is used to store notes for documents. auth_stat helps out with the author sorting and unique author statistics. doc_history is the audit trail.

There are some foreign key constraints defined for the documents table, namely that the state and class/subclass must be defined in the states and class tables. There's no constraint on codes since it can be null.

The initial sequence creation requires:

```
drop sequence doc_id_seq;
drop sequence cl_id_seq;
drop sequence ld_id_seq;
drop sequence dh_id_seq;
drop synonym members;

drop table documents;
drop table states;
drop table codes;
drop table class;
drop table class_type;
drop table doc_history;
drop table long_data;
drop table authors;
drop table auth_stat;
```

documents **Table**

```
create table documents (
   doc_id          number not null,
   doc_in_auth     varchar2(256),
   doc_non_in_auth varchar2(256),
   doc_sortkey     varchar2(60),
   doc_title       varchar2(256) not null,
   doc_description varchar2(256),
   doc_keywords    varchar2(512),
   doc_revision    varchar2(10),
   doc_published   date,
   doc_url         varchar2(256),
   doc_st_id       number not null,
   doc_public      number(1),
   doc_cd_id       number,
   doc_cl_id       number not null,
   doc_notes       number,
   doc_lastupdate  date,
   doc_updater     number,
   primary key (doc_id)
);
```

states **Table**

```
create table states (
   st_id      number,
   st_desc    varchar2(20),
   primary key (st_id)
);
```

codes **Table**

```
create table codes (
   cd_id      number,
   cd_code    char,
   cd_desc    varchar2(30),
   primary key (cd_id)
);
```

class **Table**

```
create table class (
   cl_id           number,
   cl_class        char,
   cl_subclass     char,
   cl_subname      varchar2(30),
   primary key (cl_id)
);
```

class_type **Table**

```
create table class_type (
   ct_class          char,
   ct_name           varchar2(30),
   primary key (ct_class)
);
```

doc_history Table

```
create table doc_history (
   dh_id             number not null,
   dh_doc_id         number not null,
   dh_st_id          number,
   dh_type           char,
   dh_timestamp      date not null,
   primary key (dh_id)
);
```

long_data **Table**

```
create table long_data (
   ld_id      number not null,
   ld_data    long,
   primary key (ld_id)
);
```

authors **and** auth_stat **Tables**

```
create table authors (
   au_login  varchar2(10) not null,
   au_lname  varchar2(30),
   au_fname  varchar2(30),
   primary key (au_login)
);

create table auth_stat (
   as_name    varchar2(40) not null,
   as_doc_in  number not null,
   as_isin    number,
   primary key (as_name)
);
```

Synonyms, Foreign Keys, and Sequences

```
create synonym members for resources.members;
alter table documents
 add (foreign key (doc_st_id) references states(st_id) on delete cascade,
  foreign key (doc_cl_id) references class(cl_id) on delete cascade,
  foreign key (doc_notes) references long_data(ld_id) on delete cascade,
  foreign key (doc_updater) references resources.members(mb_em_id)
 );
alter table class
 add (foreign key (cl_class) references class_type(ct_class) on delete cascade
 );

create sequence doc_id_seq;
grant select on doc_id_seq to repo_role;
create sequence cl_id_seq;
grant select on cl_id_seq to repo_role;
create sequence ld_id_seq;
grant select on ld_id_seq to repo_role;
create sequence dh_id_seq;
grant select on dh_id_seq to repo_role;
```

Triggers

```
CREATE or REPLACE trigger doc_history
BEFORE insert or update or delete on repo.documents
FOR EACH ROW

BEGIN
```

```
IF DELETING THEN
INSERT INTO repo.doc_history (
       dh_id, dh_doc_id, dh_st_id, dh_type, dh_timestamp
)
VALUES (
       dh_id_seq.nextval, :old.doc_id, :old.doc_st_id, 'D', SYSDATE
);
END IF;

IF UPDATING THEN
INSERT INTO repo.doc_history (
       dh_id, dh_doc_id, dh_st_id, dh_type, dh_timestamp
)
VALUES (
       dh_id_seq.nextval, :new.doc_id, :new.doc_st_id, 'U', SYSDATE
);
END IF;

IF INSERTING THEN
INSERT INTO repo.doc_history (
       dh_id, dh_doc_id, dh_st_id, dh_type, dh_timestamp
)
VALUES (
       dh_id_seq.nextval, :new.doc_id, :new.doc_st_id, 'I', SYSDATE
);
END IF;

END doc_history;
```

DryDock: A Document Firewall

Deepak Giridharagopal – The University of Texas at Austin

ABSTRACT

Auditing a web site's content is an arduous task. For any given page on a web server, system administrators are often ill-equipped to determine who created the document, why it's being served, how long it's been publicly viewable, and how it's changed over time.

To police our web site, we created a secure web publishing application, DryDock, that governs the replication of content from an internal, developmental web server to a stripped-down, external, production web server. DryDock codifies a formal approval process that forces management to approve all web site changes before they are pushed out to the external machine. Users never interact directly with the production machine; DryDock updates the production server on their behalf. This allows administrators to operate their production web server in a more secure and regimented network environment than normally feasible.

DryDock audits documents, tracks revisions, and notifies users of changes via email. Managers can approve files for publication at their leisure without the risk of inappropriate content ever being publicly visible. Web authors can develop pages without intimate knowledge of security policies. And administrators can instantly know the complete history of any file that has ever been published.

Introduction

Information is valuable. While nearly all organizations go to great expense to protect their networks, a much smaller percentage have formal safeguards against the accidental dissemination of sensitive information. In a web server environment where many users can update different parts of a web site at once, auditing the server for inappropriate content becomes an increasingly difficult system administration task. Most system administrators can't easily determine why a particular file exists on a web server, or who in management authorized its publication. How can administrators be expected to safeguard information if they can't tell which documents are fit for the public and which ones aren't?

This was the situation in our organization, Applied Research Laboratories, The University of Texas at Austin (ARL:UT).

Motivation

Since 1994, ARL:UT has had a publicly accessible web server. Initially, we served simple, static pages. There was relatively little web server traffic, and, like many organizations at the time, we didn't concentrate on the security of our network or our information. Our web server resided on our internal network with full access to our file server and other intranet resources, and employees were trusted to only serve documents that were appropriate for public viewing.

By 2001, our web presence had grown in both traffic and size by several orders of magnitude. Our site's much larger scale made it impossible for administrators to effectively police its content. Our formal publishing policy and guidelines were conceived for paper documents, not web pages. Their checks and balances were inadequate when applied to our web architecture, which allowed users to publish pages without even a cursory review. Many staff members were unaware that web pages even fell under these guidelines. We found ourselves unable to track who published specific files and who had deemed those files fit for the public. Since much of ARL:UT's research is sensitive and proprietary, we needed to strictly regulate the flow of information from inside our organization onto our public web server.

To ensure that only material suitable for public viewing appeared on our web site, we needed to force documents to undergo an approval process – only files that successfully complete the process will move to the web server. Furthermore, for any publicly viewable file, we needed a way to determine who authorized its publication, when the file was published, and for what reason. Not only did we need this information available for currently published files, but for any previous versions as well. A web publishing system that provided us with these features would enforce our information security policies by ensuring publicly viewable content is acceptable and well accounted for.

Due Diligence

In late 2001, we searched for tools that would put our new web publishing plan into service. Of the countless managed web publishing solutions on the market at the time, we found none that, out-of-the-box:

- implemented a role-based approval process appropriate for our organization's managerial structure
- gave us the thorough revisioning and auditing capabilities we needed

- didn't mix approved and unapproved content on our public web server
- had a friendly, web-based user interface that managers could understand
- were minimally intrusive for both our system administrators and our web developers
- used technologies we were already familiar with

Sweeping, monolithic content management systems such as Vignette StoryServer [17] were inappropriate for our environment. ARL:UT is comprised of many autonomous groups that have their own web development methods and practices. While the need for a formal information security process was universally recognized, a sweeping content management system was both politically infeasible and far too expensive.

Portal and weblog systems such as PostNuke [7], Tiki [20], or Plone [8] focus on creating highly dynamic, interactive web sites. Thus, they frequently offer collaborative features such as article syndication, Wiki[1] systems, forums, and user commentary. At ARL:UT, however, we needed a strict web presence that was decidedly static and non-collaborative – this obviated many of these systems' features. All we wanted was software that would allow approved documents through to the web server, while blocking all other unauthorized updates. All of these packages required so much customization that it was easier for us to build our own solution, tailored specifically to our environment.

Having resigned ourselves to writing a custom tool, we began looking at platforms upon which we could base our application. We were particularly interested in the Zope [16] application server. Written in Python, Zope has been used to build many complex and dynamic web sites. Its features include user management, web-based administration, searching, clustering, and syndication. Like the aforementioned packages, however, a great deal of Zope's dynamic componentry was of no use to us, and much of Zope's functionality fell far outside the scope of simple publication oversight. Though these issues weren't intractable, when combined with Zope's steep learning curve, they led us to look at other less complex and less ambitious platforms.

We settled on WebKit, the Webware for Python [22] application server. WebKit uses a design pattern

[1] "Wiki Wiki Web is a set of pages of information that are open and free for anyone to edit as they wish, through a web interface. The system creates cross-reference hyperlinks between pages automatically. Anyone can change, delete, or add to anything they see." [10]

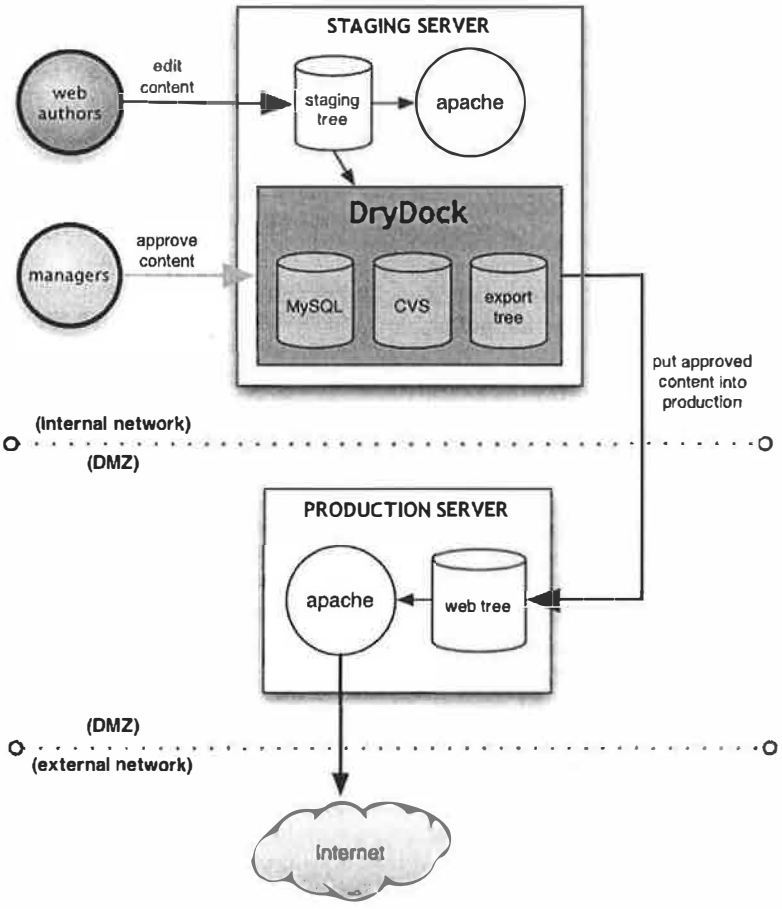


Figure 1: Web publishing with DryDock.

fashioned after Sun's Java Servlet [15] architecture (a paradigm we were familiar with), and includes little extraneous, dynamic componentry we'd have to work around. We could implement all of the heavy-lifting functionality in plain Python and use a small number of servlets to expose a web interface – we found such an architecture much more workable via WebKit than Zope.

Using WebKit, we devised an application that would give us the security and auditing features we required. Several months later, we put DryDock into production. DryDock has been managing our web publishing for over a year and a half now.

What Is DryDock?

DryDock tackles our information security problem in two main ways: by implementing a dual web server setup, and by forcing a separation between creating content and approving its publication.

Figure 1 details the process of web publishing using DryDock. DryDock's dual web server setup is comprised of a *production* machine and a *staging* machine, both of which have identical web server configurations. The production server holds content suitable for the public and resides in a publicly accessible DMZ[2] [14], while the staging server resides behind the firewall as part of the internal network.

The staging server houses a web tree containing files under development (the *development* tree) and a separate tree containing files authorized for publication (the *export* tree). The development tree is

[2]Demilitarized Zone: a "neutral zone" between a company's private network and the outside public network.

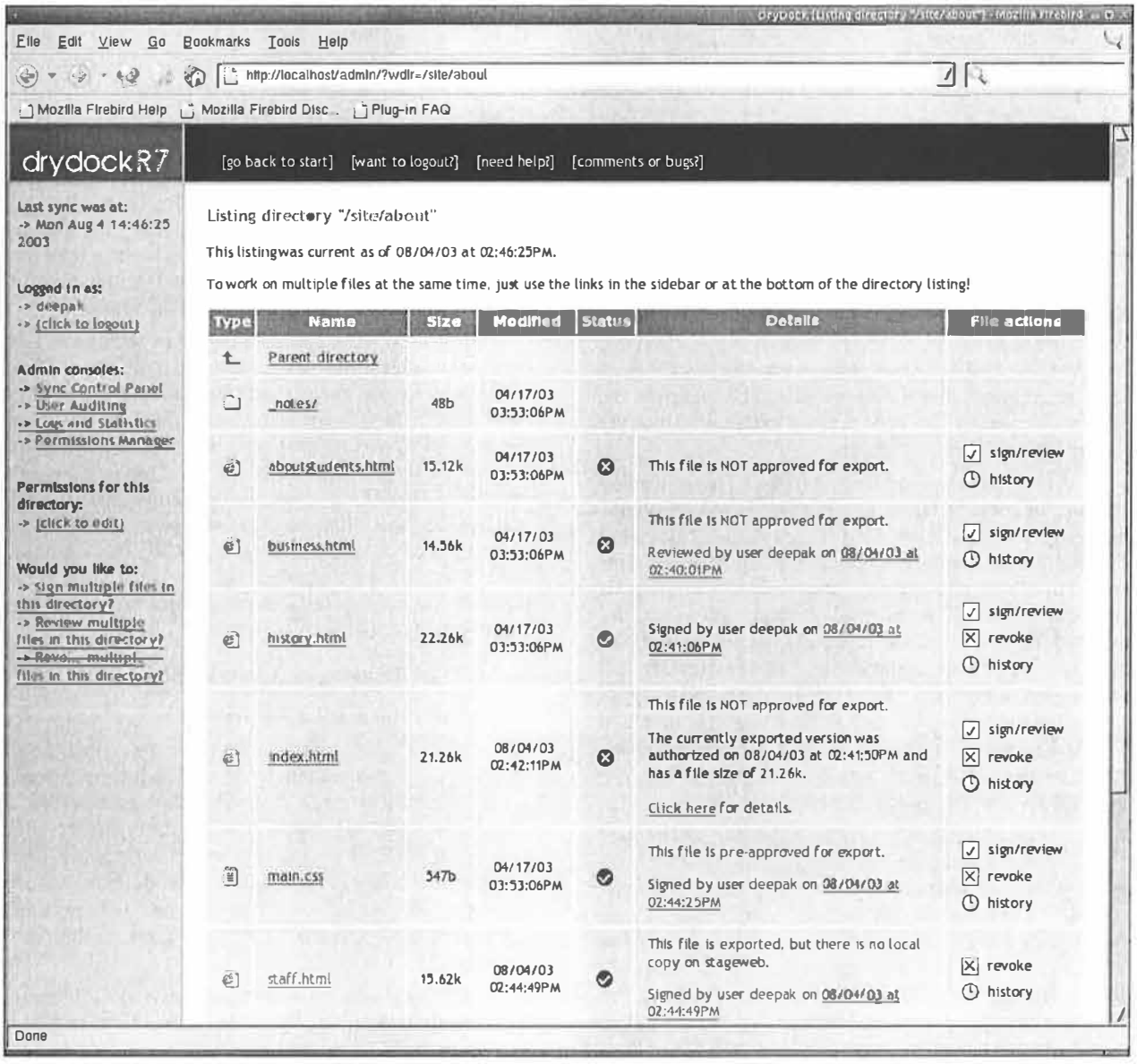


Figure 2: The DryDock directory listing screen.

accessible to web authors on the internal network, while the export tree is solely managed by DryDock.

Managers (*content approvers*) use DryDock's web interface (Figure 2) to browse the staging server. The web interface presents an integrated and easily discernible view of the development tree and the export tree which shows them:

- which files from the development tree are authorized for publication
- which files still await approval
- the differences between development and export versions of the same file
- any file's approval history

After looking over an unapproved file and finding it fit for public viewing, a content approver *signs* the file, instructing DryDock to mark the file as authorized for publication. For added security, DryDock can be configured to require additional information to complete the signing process, such as the name of the document author or a note explaining why the file is being signed.

Once the file is signed, DryDock automatically copies it from the development tree to the export tree. Finally, DryDock synchronizes the staging server's export tree to the production server over SSH, replacing the external web site with an updated copy containing newly signed files.

Since the staging server's export tree is separate from the development tree, deleting a file from one will not delete it from the other. To remove a file from the export tree, a content approver uses DryDock's web interface to mark that file as *revoked*. This removes the file from the set of files copied to the external web server during the next synchronization.

While signing authority is given to a limited number of users, a larger number can be given *review* authority, or the ability to *soft-sign* files. Reviewing files follows the same process as signing files, save for two important differences: approval information is optional and can be incomplete, and reviewed files still require an authorized signature for publication. At ARL:UT, review is employed by users to partially fill in approval information for a file so that when a signer moves to authorize that file for publication, much of the data required is already present. Review provides much of the same functionality as signing a file, but without a signature's consequences.

Though DryDock's approval process works well for most documents, rapidly changing documents must be continually re-signed by content approvers for each change to appear on the external web site. To ease dealing with these types of files, a user can sign them as *pre-approved*; files marked as such will be copied to the production server during synchronization even if their contents have changed since they were signed. Since file pre-approval circumvents DryDock's typical workflow process, we advise our users to apply the option sparingly; users must be vigilant about pre-approved files' content.

Web Developer Migration

Migrating web developers to DryDock's web publishing process should be painless. Instead of placing content directly onto the production server, web developers now place documents onto the staging server. At ARL:UT, we configured our staging server to support the same access methods our web developers have always used: FTP, Samba, and NFS. Since nearly all off-the-shelf web development tools can use at least one these methods to edit pages, we simply instructed our web authors to re-configure their tools to point to the appropriate directory on the staging server.

In most cases, web developers shouldn't have to change their pages for DryDock. In a proper DryDock setup, the staging server's web server environment mimics that of the production server, so properly functioning pages on the staging server will still work when exported to the production machine.

The lone caveat is the avoidance of absolute links: since the production server (not the staging server) will be serving up content to the public, any links in a web page explicitly mentioning the staging server will not work. Relative links solve this problem by not mentioning the server name in the link address; the web server assumes the document is local.

However, there is one situation in which use of a relative link is impossible. If page authors need to refer to a secure page from an insecure page, they need to refer to the host using https:// instead of http://. Since it isn't possible to specify a protocol in a relative link, an absolute link is required. The solution we employed at ARL:UT was to use a simple server-side-include variable that contains the host name of the machine from which the page is served. Page authors then construct an absolute link, using the variable in place of the host name, and the correct server name is inserted dynamically.

Benefits to Administrators

Improved Information Security

DryDock safeguards against the dissemination of inappropriate web content by codifying a formal document approval process. It ensures that all updates to web content are inspected for propriety *before* they ever escape the shelter of the internal network. DryDock makes users accountable for the documents they approve; if content is found to be inappropriate, administrators can easily determine who let that content through.

DryDock acts as a *document firewall*. Just as a traditional firewall regulates the flow of packets to a private network, DryDock regulates the flow of documents to the production web server; they are both

access control mechanisms. Through signing, pre-approving, and revoking files, DryDock users create rule sets that manage the movement of documents into and out of its export tree.

Content Auditing

DryDock goes to great lengths to log all user and system activity and keep administrators abreast of all changes in web content. DryDock provides administrators with an auditing console that lets them browse DryDock's records. Administrators can inspect the most recent events, all events pertaining to a user, or any events within a specified date range, or they can perform a free-text search to tailor the results to their liking. For greater detail, administrators can peruse DryDock's extensive log files. To immediately notify administrators of changes in content, DryDock can be configured to send out email whenever activity occurs.

With these tools, administrators can easily diagnose problems with content. If a document is approved that is later found to be unsuitable for the public, administrators can refer to the email DryDock sent when the file was signed; the email indicates the parties responsible for approving the file and the file's full path. Administrators can revoke the document's publication or, if necessary, manually restore a previous version of the file using DryDock's revisioning system. They then can use the auditing console to find any other files upon which the responsible parties have recently operated (Figure 3), and handle them as needed.

Improved Web Server Security

DryDock's dual web server setup makes the production server easy to harden. Users never interact directly with the production machine; DryDock interacts with it on their behalf. This allows administrators to operate the production server in a more secure and regimented network environment than normally feasible. Because normal users never access the web server, system administrators can restrict its logins to a single administrative account. As DryDock updates the server's content over SSH, traditional file access services such as Samba, NFS, or FTP can also be disabled.

If the web site is comprised of mostly static pages and simple scripts, backups are much less

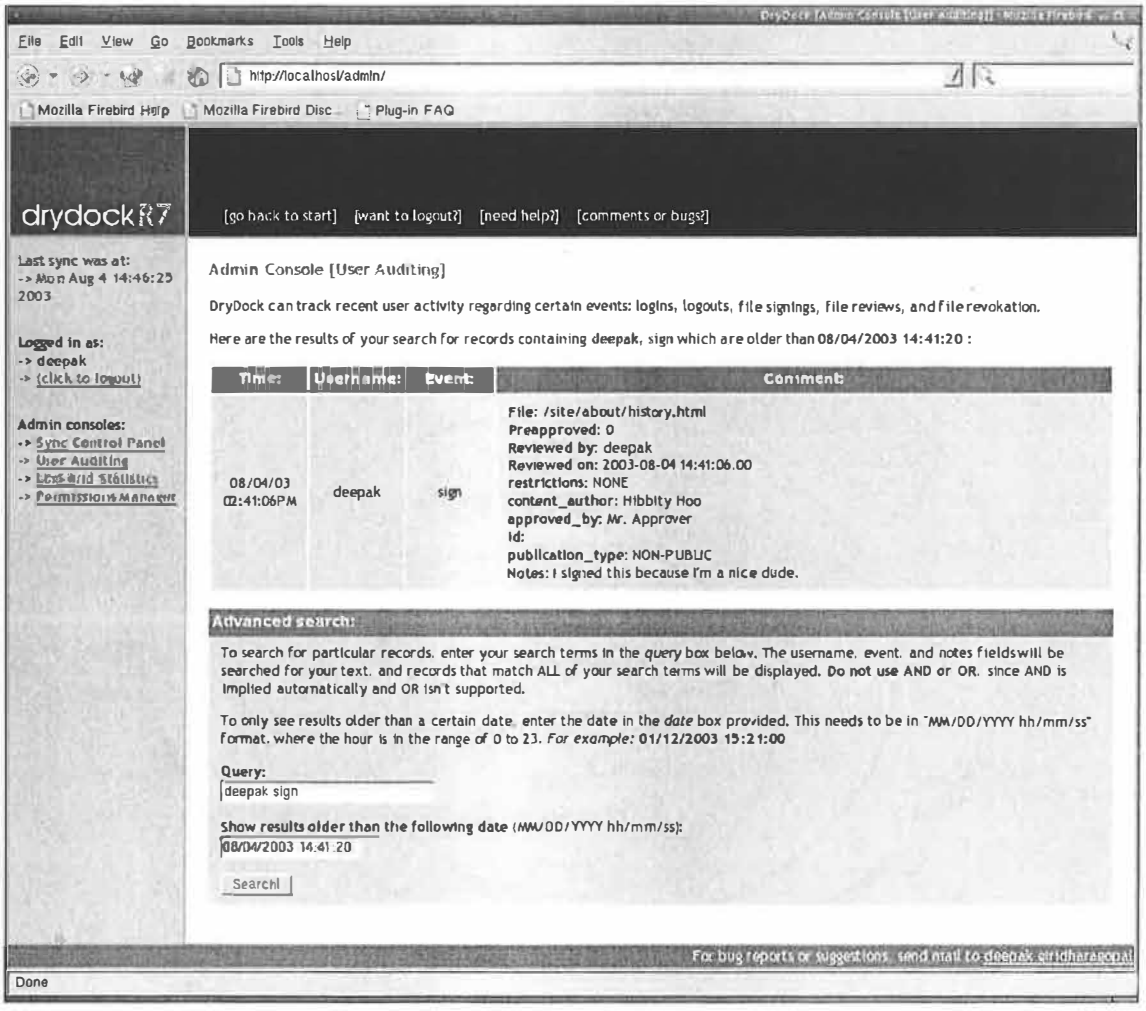


Figure 3: Auditing console.

complicated. Since DryDock rebuilds the entire web tree during each synchronization, there's no need to maintain backups of the production web server beyond a base image of the machine's initial configuration. DryDock simply treats a production web server as a drone – all web content is housed safely inside the firewall on the staging server, and it's pushed out to the external machine when necessary.

Finally, DryDock's repeated rebuilding of the production web tree impedes a naïve intruder from simply defacing web pages. The corrupted documents will simply be reconstituted during the next synchronization.

DryDock Composition

DryDock is written in Python and uses WebKit for request handling and session management. The user interface is written in HTML using Cheetah [2], a Python templating library. DryDock's back-end is comprised of four main components: a relational database that stores auditing information, a role-based permissions system, a revisioning system that tracks changes in approved documents, and a synchronization daemon that updates the production web server.

The Database

Since DryDock needs to query its data on a per-file, per-directory, and per-user basis, storing the information in a relational database was a natural fit. DryDock uses MySQL [9] to store authorization information for files, permission definitions for users, review information, and user activity logs.

Each time a file is signed, reviewed, or revoked, DryDock records the operation's details in its tables. DryDock remembers the user, the time, the file's current MD5 fingerprint, users' notes, and any additional information DryDock has been configured to accept. DryDock uses this data to display a file's transaction history and to determine if a file's contents have changed since it was last signed or reviewed.

Permissions

DryDock features a role-based permissions model. Users and groups from the underlying UNIX system can be assigned a role of *admin*, *sign*, *review*, *view*, or *none* per path on the staging web tree. A role circumscribes all of the actions a user can perform; any capabilities not specifically permitted are prohibited for that directory and all paths underneath. Figure 4 describes the different roles and shows how they are cumulative in design; for example, a user with *sign* authority for a path also has *review* authority.

Permissions for a user resolve in a bottom-up manner. If no role is defined for the user on a path, DryDock searches for one defined for the user on the path's parent directory. The process continues until a role is found or the root directory is reached. If no role is found for the user, then DryDock performs the same recursive check for each group the user is in. If there is still no matching role, DryDock assumes the user has no privileges for the initial path.

The Revisioning System

To let administrators see a document's evolution, DryDock relies on the freely available Concurrent

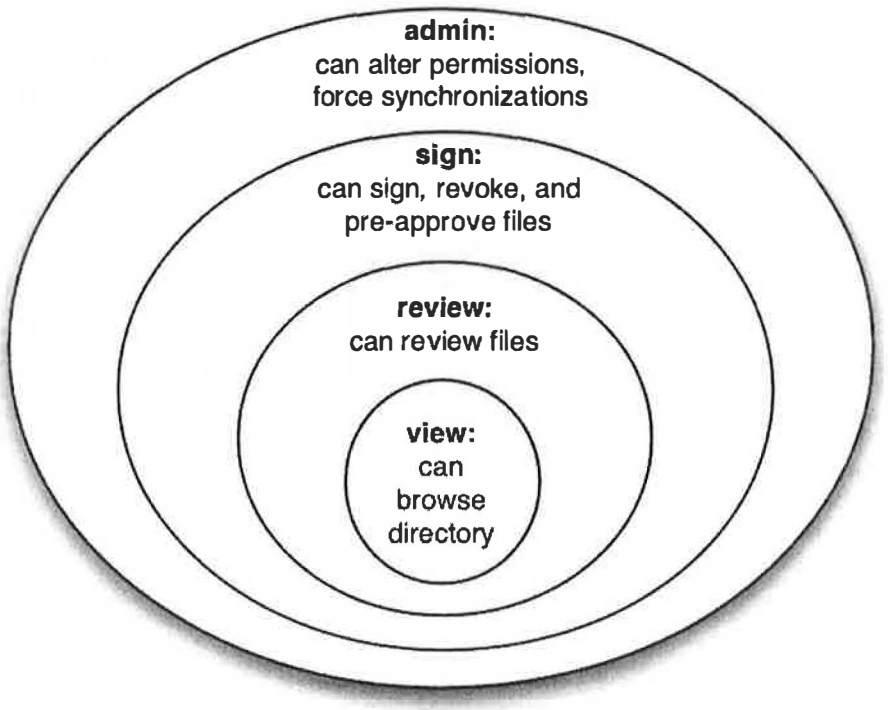


Figure 4: Role-based permissions model.

Versions System [1] (CVS) to track changes in approved files. CVS provides stable, production quality, multi-file version control. DryDock interacts with CVS by coordinating files among three directories during versioning operations: the staging tree, a CVS repository ("a complete copy of all the files and directories which are under version control" [6]), and a CVS working directory (a working copy of the repository used to commit changes to the repository).

When a file is signed, DryDock copies the file to the CVS working directory and then commits it to the repository. Similarly, when a file is revoked, DryDock deletes the file from the CVS working directory and then notifies the repository that the file has been removed. Through use of these mechanisms, the CVS working directory always contains the current version of each approved file.

While CVS adequately handled most of our revisioning needs, its file-based design couldn't handle changes in the repository's directory structure. "Because it uses the RCS storage system under the hood, CVS can only track file contents, not tree structures" [3]. CVS provides no way to remove a directory from the repository without losing all of its versioning information. So, if you delete all the files within a directory and still want to access those files' revision histories, the directory must remain in the repository. This lingering directory prohibits adding a new file with the same name to the repository because UNIX file systems won't allow two identically named items in a directory. If a file cannot be added to the repository, then DryDock users cannot approve that file.

To remedy this, we distort directories' names when they are added to the working directory. Whenever DryDock creates a directory in the CVS working directory, it prepends a predetermined sequence of characters to the directory's name; this mangled name is used when the directory is added to the repository. Concurrently, we prohibit users from creating files beginning with the same reserved characters. This allows us to have directories and files with the same name under version control simultaneously.

Synchronization

With DryDock, users never make updates directly to the production web server. Synchronization (*sync*), DryDock's process of pushing documents out to the production machine's web tree, is the sole way to update the external web site. Periodically, DryDock copies approved versions of all documents to the external machine, reconstructing the production web site.

Since synchronizations are the only way changes propagate to the production machine, we needed to schedule them frequently. Instructing DryDock to immediately sync whenever a user signed or revoked a file would work well in periods of light use, but the staging server would be overwhelmed if users signed and revoked pages en masse.

Our solution was to employ *delayed syncs*. Instead of immediately syncing when a user signs or revokes a file, DryDock schedules a sync to occur five minutes later. If users sign or revoke additional files inside this five-minute window, DryDock reschedules the sync for five minutes from the time of the most recent approval operation. This process continues until there is no activity for the duration of the window, at which time the sync occurs. Since heavy usage would keep pushing the sync back by five minutes, perhaps indefinitely, we instituted a one-hour failsafe between syncs. If a sync hasn't occurred in the last 60 minutes, one is forced. Grouping updates in this way gave us a reasonable compromise between update frequency and server load. For situations requiring finer control, however, we allow DryDock administrators to force a sync on demand.

Figure 5 details the sync process. Sync is split into two parts: handling pre-approved files and exporting signed files to the production machine.

Pre-approved files require special handling during sync to ensure their changes are monitored by

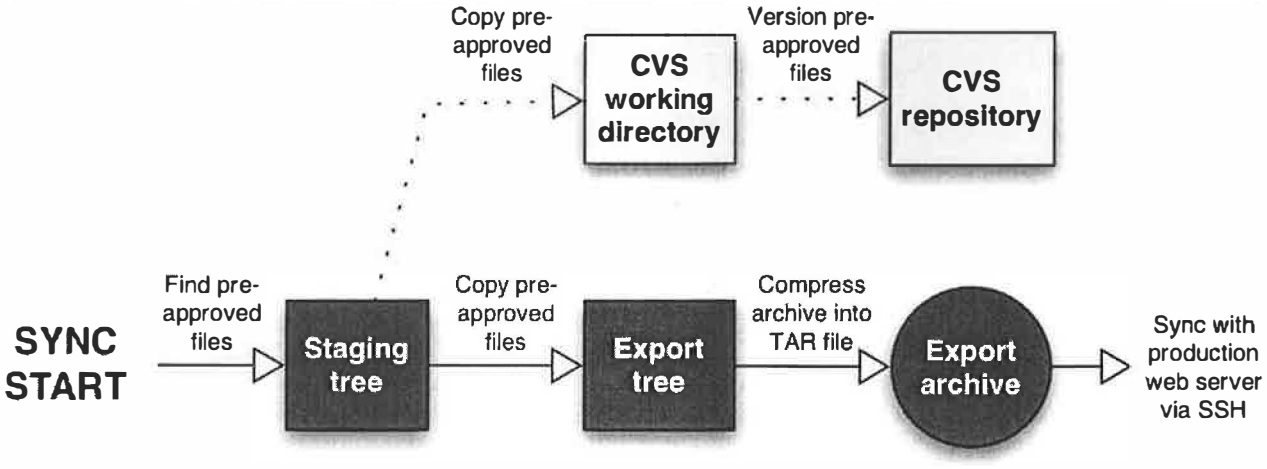


Figure 5: Synchronization flow.

DryDock's revisioning system. Ordinarily, DryDock only adds files to its revisioning system when they are signed or revoked. Since users aren't required to sign pre-approved files each time their contents change, DryDock would normally be unable to track changes in pre-approved documents over time. To remedy this, DryDock adds the current state of every pre-approved file to its revisioning system at the start of each sync.

Before DryDock exports approved files to the external web server, it must construct an image of the production web tree. Originally, we used CVS's export ability. A CVS export would copy the most recent versions of all active files to a temporary directory. Since the CVS repository contains the current versions of all approved documents, this temporary directory would, by definition, contain an image of the production web tree. We then normalized any "mangled" directory names, converted the directory to a tar archive, and copied the file to the external machine.

Action	Time	% of Total
Copy pre-approved files	1s	1.0%
Add pre-appr'd files to CVS	22s	21.7%
CVS export	53s	52.5%
Normalize directory names	2s	2.0%
Tar archival	7s	6.9%
Update the production server	16s	15.8%
Total	101s	

Figure 6: Performance of CVS-export-based synchronization at ARL:UT.

Though this worked correctly, synchronizations performed poorly. After investigation, we discovered that our use of CVS's export facility was a sizable performance bottleneck; the synchronization process spent over half its time waiting for the export to complete (Figure 6).

Our answer was to continually maintain an image of the production web tree in a directory alongside the CVS repository; this is the *export* directory. Whenever a user signs a file DryDock copies it to the export directory, and whenever a file is revoked DryDock deletes it from the export directory.

By continually maintaining this mirror of the production web site, we no longer need to perform a CVS export or normalize any directory names. Instead of creating this mirror at sync time, we can spread the work out over time, updating the mirror as changes occur. Table 6 shows that, by skipping these steps, we cut the sync time by 54.5% from 101 seconds to only 46 seconds.

DryDock can use a variety of user-defined scripts to transmit the tar-archived export directory to the external web server. At ARL:UT, we use SSH. The production server is configured to automatically decompress the archive and replace its current web root with the new content. To make the process more secure, we use a pair of public and private cryptographic keys to establish the connection instead of a traditional user name and password combination, and the key is associated exclusively with the specific script that updates the machine's web root.

Evaluation

All told, DryDock has moved us away from our largely unsecured and unrestricted web publishing process to one that affords us much better information security. It has made it possible for a large number of web authors to safely publish content to a web server that remains almost completely isolated from our internal network. By removing direct user interaction with the external web server, DryDock has allowed us to secure our web server to a degree not before possible [11]. And by enforcing a formal approval process, DryDock has given our management and administrators total publication oversight.

DryDock has been governing our web presence for over one and a half years, and has proven itself to be stable and dependable. This notwithstanding, there are still several areas that could be improved.

User Interface Issues

While DryDock scales well vis-a-vis directories containing thousands of files, its performance is overshadowed by slow web page rendering and the problems users have when operating on such large sets of data. DryDock uses HTML tables to display information about each file, and table rendering is notoriously taxing on web browsers. Consequently, as directories grow massive, a web browser cannot display much of DryDock's user interface quickly.

Furthermore, DryDock's web interface isn't truly designed to operate on large amounts of data. Though users can simultaneously sign, review, or revoke multiple files in the same directory, the usability of a point-and-click, forms-based web interface doesn't scale well when dealing extremely large numbers of inputs.

In spite of these structural issues, we've had few performance-related complaints. Most web developers simply don't work with thousands of files in a single directory – it's far too cumbersome. DryDock, optimized for interactive performance with modest data sets, suits the usage patterns and scale at ARL:UT.

Moving Away From CVS

While we were able to work around many of CVS's directory management problems through name mangling, overcoming its lack of a truly programmable interface is more problematic. To interact with CVS, DryDock is limited to invoking CVS's command-line tools and parsing their output. This output is designed to be easily readable by humans, not easily parsed by software [19]. This hinders DryDock's integration with

CVS, and tight integration with a revisioning system is a prerequisite for DryDock to provide repository management and inspection features.

CVS's directory management issues and its command-line interface are structural problems not likely to be remedied. We will migrate from CVS and towards a revisioning system that better meets our requirements.

Ostensibly, the next-generation version control system that will best suit our needs is Subversion [4, 18]. Subversion tracks changes in directory structure, obviating our name mangling. More importantly, however, Subversion has a well-defined C programming interface and a nascent Python interface (DryDock's native tongue). Subversion is currently in development and we await its first stable release.

File Pass-through

Since DryDock tracks changes in all approved documents, the size of DryDock's CVS repository increases as time goes on. Pre-approved files exacerbate the problem since their changes are committed to CVS at each sync; as syncs occur at least once per hour, pre-approved files are committed to CVS at least 24 times daily.

One solution to this problem is to allow users to flag signed files as *pass-through*: files marked as such will not be added to DryDock's revisioning system. A signer marks a file as pass-through if tracking its changes over time isn't important. Marking pre-approved files as pass-through can slow the repository's growth, though at the expense of thoroughness.

Incremental syncs

To reduce sync time, DryDock could transmit updates to the production machine during each synchronization instead of transmitting the entire web tree. During a sync, DryDock could query the database to determine which files have been signed or revoked since the last sync. It could use this information to construct an archive containing any recently signed files, all pre-approved files, and a text file listing which files need to be deleted from the production server. This archive could then be transmitted to an update script on the external machine that will place the updated files in the correct locations and delete any files specified in the deletion list.

For added safety, these incremental syncs can be intermingled with full syncs. For added performance with truly large web sites, DryDock could even employ a binary differences algorithm such as XDelta [21] to transfer only the parts of individual files that have changed.

Initially, it would seem appropriate to use rsync[3] [12] to reconcile and transmit the differences between the export directory and the production server's web tree. However, rsync is designed to operate over high-latency, low bandwidth links; at ARL:UT, we have a switched 100-megabit connection between the staging and production servers. Furthermore, unlike rsync, DryDock is omniscient; it knows before opening a connection to the production server which files require updating. Instead of spending connection time determining which file chunks need to be sent over, DryDock can simply transfer data.

[3] An incremental file transfer tool. "rsync uses the *rsync algorithm* [21] which provides a very fast method for bringing remote files into sync. It does this by sending just the differences in the files across the link, without requiring that both sets of files are present at one of the ends of the link beforehand." [13]

Availability

DryDock is currently available at http://www.arlut.utexas.edu/DryDock, and we plan on having placed DryDock under the GNU Public License by the time of the conference.

DryDock requires Python, Webware, CVS, SSH, and MySQL. DryDock has been designed and tested on Linux and Solaris, and it is expected to run on any modern UNIX platform that supports the aforementioned tools.

Acknowledgments

Many people have contributed to DryDock's development. Jonathan Abbey greatly assisted in DryDock's design and made important contributions to DryDock's permissions handling and synchronization routines. Dan Scott lent administrative support to the project and helped lay out many of DryDock's initial requirements, and Nanette Lemma provided invaluable user feedback.

I'd like to thank Jonathan Abbey (again) and Chad Duffy for tirelessly reading and commenting on the many iterations of this paper. I'd also like to thank Michael Gilfix, this paper's steward, for his refinements.

Lastly, I'd like to thank the authors of Webware for their wonderful work, the developers of Python for a great programming language, and Applied Research Laboratories for supporting my software development efforts, both large and small.

What's in a Name?

The name "DryDock" is derived from nautical terminology. A drydock is "a specialized dock where boats are pulled out of the water to be repaired, painted, or inspected" [5]. We found the analogy appropriate.

Author Information

Deepak Giridharagopal is the primary developer of DryDock. He holds a B.S. in computer science from The University of Texas at Austin. After working for Reactivity, Inc. doing Enterprise Java development, he now works at Applied Research Laboratories writing software for their Computer Science Division. Besides all things computer-related, he is thoroughly

entertained by automobiles, The Smiths, giant transforming robots, and breakdancing. He can be reached by email at deepak@arlut.utexas.edu.

References

[1] Berliner, Brian, "CVS II: Parallelizing Software Development," *Proceedings of the USENIX Winter 1990 Technical Conference*, USENIX, pp. 341-352, 1990.

[2] *Cheetah – the python powered template engine*, http://www.cheetahtemplate.org, 2003.

[3] Collins-Sussman, Ben, "The Subversion Project: Building a Better CVS," *Linux Journal*, Num. 94, 2002.

[4] Collins-Sussman, Ben, Brian Fitzpatrick, and C. Michael Pilato, *Subversion: The Definitive Guide*, http://svnbook.red-bean.com/html-chunk/, 2003.

[5] *sitesalive Glossary*, http://www.sitesalive.com/admin/glossary/sectD.html, July, 2003.

[6] Fogel, Karl Franz, *Open Source Development with CVS*, The Coriolis Group, 1999.

[7] Haakenson, Vanessa, *What is postnuke?*, http://noc.postnuke.com/docman/view.php/5/12/whatispostnuke.htm, June, 2003.

[8] McKay, Andy and Amr Malik, *The Plone Book*, http://www.plone.org/documentation/book, June, 2003.

[9] *Mysql: An Open Source Relational Database*, http://www.mysql.org, 2003,

[10] *O'Reilly network: Wiki Wiki Web*, http://www.oreillynet.com/pub/d/282, 2003,

[11] Rhodes, Charles, "The Internal Threat to Security or Users Can Really Mess Things Up," *GSEC Practical*, http://www.sans.org/rr/papers/8/856.pdf, 2003.

[12] *Rsync*, http://rsync.samba.org, June, 2003.

[13] *Rsync Readme*, http://rsync.samba.org/README.html, 2003.

[14] Runnels, G. Michael, "Implementing Defense in Depth at the University Level," *GSEC Practical*, Num. 1.4, http://www.sans.org/rr/paper.php?id=596, 2002.

[15] *Java Servlet Technology*, http://java.sun.com/products/servlet/, July, 2003,

[16] Spicklemire, Steve, Kevin Friedly, Jerry Spicklemire, and Kim Brand, "Zope: Web Application Development and Content Management," *Que*, 2001.

[17] *Vignette – Content Management and Portal Solutions*, http://www.vignette.com, July, 2003

[18] *Subversion: A Compelling Replacement for CVS* http://subversion.tigris.org, June, 2003.

[19] Taler, Alexander, *libCVS*, http://libcvs.cvshome.org/servlets/ProjectHome, June, 2003.

[20] *Tikiwiki*, http://tikiwiki.sourceforge.net, 2003.

[21] Tridgell, Andrew and Paul Mackerras, "The rsync Algorithm," *Tech. Report TR-CS-96-05*, Australian National University, June, 1996.

[22] *Webware for python*, http://webware.sourceforge.net, 2003.

Run-time Detection of Heap-based Overflows

William Robertson, Christopher Kruegel, Darren Mutz, and Fredrik Valeur
– University of California, Santa Barbara

ABSTRACT

Buffer overflows belong to the most common class of attacks on today's Internet. Although stack-based variants are still by far more frequent and well-understood, heap-based overflows have recently gained more attention. Several real-world exploits have been published that corrupt heap management information and allow arbitrary code execution with the privileges of the victim process.

This paper presents a technique that protects the heap management information and allows for run-time detection of heap-based overflows. We discuss the structure of these attacks and our proposed detection scheme that has been implemented as a patch to the GNU Lib C. We report the results of our experiments, which demonstrate the detection effectiveness and performance impact of our approach. In addition, we discuss different mechanisms to deploy the memory protection.

Introduction

Buffer overflow exploits belong to the most feared class of attacks on today's Internet. Since buffer overflow techniques have reached a broader audience, in part due to the Morris Internet worm [1] and the Phrack article by AlephOne [2], new vulnerabilities are being discovered and exploited on a regular basis. A recent survey [3] confirms that about 50% of vulnerabilities reported to CERT are buffer overflow related.

The most common type of buffer overflow attack is based on stack corruption. This variant exploits the fact that the return addresses for procedure calls are stored together with local variables on the program's stack. Overflowing a local variable can thus overwrite a return address, redirecting program flow when the function returns. This potentially allows a malicious user to execute arbitrary code.

Recently, however, buffer overflows that corrupt the heap have gained more attention. Several CERT advisories [4,5] describe exploits that affect widely deployed programs. Heap-based overflows can be divided into two classes: One class [6] comprises attacks where the overflow of a buffer allocated on the heap directly alters the content of an adjacent memory block. The other class [7,8] comprises exploits that alter management information used by the memory manager (i.e., malloc and free functions). Most malloc implementations share the behavior of storing management information within the heap space itself. The central idea of the attack is to modify the management information in a way that will allow subsequent arbitrary memory overwrites. In this way, return addresses, linkage tables or application level data can be altered. Such an attack was first demonstrated by Solar Designer [9].

This paper introduces a technique that protects the management information of boundary-tag-based heap managers against malicious or accidental modification. The idea has been implemented in Doug Lea's malloc for GNU Lib C, version 2.3 [10], utilized by Linux and Hurd. It could, however, be easily extended to other systems such as various free BSD distributions. Using our modified C library, programs are protected against attacks that attempt to tamper with heap management information. It also helps to detect programming errors that accidentally overwrite memory chunks, although not as complete and verbose as available memory debuggers. Program recompilation is not required to enable this protection. Every application that is dynamically linked against Lib C is secured once our patch has been applied.

Related Work

Much research has been done on the prevention and detection of stack-based overflows. A well-known result is StackGuard [11], a compiler extension that inserts a 'canary' word before each function return address on the stack. When executing a stack-based attack, the intruder attempts to overflow a local buffer allocated on the stack to alter the return address of the function that is currently executing. This might permit the attacker to redirect the flow of execution and take control of the running process. By inserting a canary word between the return address and the local variables, overflows that extend into the return address will also change this canary and thus, can be detected.

There are different mechanisms to prevent an attacker from simply including the canary word in his overflow and rendering the protection ineffective. One solution is to choose a random canary value on process startup (i.e., on exec) that is infeasible to guess. Another solution uses a terminator canary that consists of four different bytes commonly utilized as string terminator characters in string manipulation library

functions (such as strcpy). The idea is that the attacker is required to insert these characters in the string used to overflow the buffer to overwrite the canary and remain undetected. However, the string manipulation functions will stop when encountering a terminator character and thus, the return address remains intact.

A similar idea is realized by StackShield [12]. Instead of inserting the canary into the stack, however, a second stack is kept that only stores copies of the return addresses. Before a procedure returns, the copy is compared to the original and any deviations lead to the abortion of the process. Stack-based overflows exploit the fact that management information (the function return address) and data (automatic variables and buffers) are stored together. StackGuard and StackShield are both approaches to enforcing the integrity of in-band management information on the stack. Our technique builds upon this idea and extends the protection to management information in the heap.

Other solutions to prevent stack-based overflows are not enforced by the compiler but implemented as libraries. Libsafe and Libverify [13,14] implement and override unsafe functions of the C library (such as strcpy, fscanf, getwd). The safe versions estimate a safe boundary for buffers on the stack at run-time and check this boundary before any write to a buffer is permitted. This prevents user input from overwriting the function return address.

Another possibility is to make the stack segment non-executable [15]. Although this does not protect against the actual overflow and the modification of the return address, the solution is based on the observation that many exploits execute their malicious payload directly on the stack. This approach has the problem of potentially breaking legitimate uses such as functional programming languages that generate code during run-time and execute it on the stack. Also, gcc uses executable stacks as function trampolines for nested functions and Linux uses executable user stacks for signal handling. The solution to this problem is to detect legitimate uses and dynamically re-enable execution. However, this opens a window of vulnerability and is hard to do in a general way.

Less work has been done on protecting heap memory. Non-executable heap extensions [16, 17] that operate similar to their non-executable stack cousins have been proposed. However, they do not prevent buffer overflows from occurring and an attacker can still modify heap management information or overwrite function pointers. They also suffer from breaking applications that dynamically generate and execute code in the heap.

Systems that provide memory protection are memory debuggers, such as Valgrind [18] or Electric Fence [19]. These tools supervise memory access (read and write) and intercept memory management calls (e.g., malloc) to detect errors. These tools use an approach similar to ours in that they attempt to maintain the integrity of the utilized memory. However, a check is inserted on every memory access, while our approach only performs a check when allocating or deallocating memory chunks. Memory debuggers effectively prevent unauthorized memory access and stop heap-based buffer overflows. Yet, they also impose a serious performance penalty on the monitored programs, which often run an order of magnitude slower. This is not acceptable for most production systems.

A recent posting on bugtraq pointed to an article [20] that discusses several techniques to protect stack and heap memory against overflows. The presented heap protection mechanism follows similar ideas as our work as it aims at protecting heap management information. However, no details were provided and no implementation or evaluation of their technique exists.

A possibility of preventing stack-based and heap-based overflows altogether is the use of type-safe languages such as Java. Alternatively, solutions have been proposed [21] that provide safe pointers for C. All these systems can only be attacked by exploiting vulnerabilities [22, 23] in the mechanisms that enforce the type safety (e.g., bytecode verifier). Note, however, that safe C systems typically require new compilers and recompilation of all applications to be protected.

Technique

Heap Management in GNU Lib C (glibc)

The C programming language provides no built-in facilities for performing common operations such as dynamic memory management, string manipulation or input/output. Instead, these facilities are defined in a standard library, which is compiled and linked with user applications. The GNU C library [10] is such a library that defines all library functions specified by the ISO C standard [24], as well as additional features specific to POSIX [25] and extensions specific to the GNU system [26].

Two kinds of memory allocation, static and automatic, are directly supported by the C programming language. Static allocation is used when a variable is declared as static or global. Each static or global variable defines one block of space of a fixed size. The space is allocated once, on program startup as part of the exec operation and is never freed. Automatic allocation is used for automatic variables such as a function arguments or local variables. The space for an automatic variable is automatically allocated on the stack when the compound statement containing the declaration is entered, and is freed when that compound statement is exited.

A third important kind of memory allocation, dynamic allocation, is not supported by C variables but is available via glibc functions. Dynamic memory allocation is a technique in which programs determine during run-time where information should be stored. It

is needed when the amount of required memory or when the lifecycle of memory usage depends on factors that are not known a-priori. The two basic functions provided are one to dynamically allocate a block of memory (malloc), and one to return a previously allocated block to the system (free). Other routines (such as calloc, realloc) are then implemented on top of these two procedures.

GNU Lib C uses Doug Lea's memory allocator dlmalloc [27] to implement the dynamic memory allocation functions. dlmalloc utilizes two core features, boundary tags and binning, to manage memory requests and releases on behalf of user programs.

Memory management is based on 'chunks,' memory blocks that consist of application usable regions and additional in-band management information. The in-band information, also called boundary tag, is stored at the beginning of each chunk and holds the sizes of the current and the previous chunk. This allows for coalescing two bordering unused chunks into one larger chunk, minimizing the number of unusable small chunks as a result of fragmentation. Also, all chunks can be traversed starting from any known chunk in either a forward or backward direction.

Chunks that are currently not in use by the application (i.e., free chunks) are maintained in bins, grouped by size. Bins for sizes less than 512 bytes each hold chunks of only exactly one size; for sizes equal to or greater than 512 bytes, the size ranges are approximately logarithmically increasing. Searches for available chunks are processed in smallest-first, best-fit order, starting at the appropriate bin depending on the memory size requested. For unallocated chunks, the management information (boundary tag) includes two pointers for storing the chunk in a double linked list (called free list) associated with each bin. These list pointers are called forward (fd) and back (bk).

On 32-bit architectures, the management information always contains two 4-byte size-information fields (the chunk size and the previous chunk size). When the chunk is unallocated, it also contains two 4-byte pointers that are utilized to manipulate the double linked list of free chunks for the binning.

This basic algorithm is known to be very efficient. Although it is based upon a search mechanism to find best fits, the use of indexing techniques (i.e., binning) and the exploitation of special cases lead to average cases requiring only a few dozen instructions, depending on the machine and the allocation pattern. A number of heuristic improvements have also been incorporated into the memory management algorithm in addition to the main techniques. These include locality preservation, wilderness preservation, memory mapping, and caching [28].

Anatomy of a Heap Overflow Exploit

The use of in-band forward and back pointers to link available chunks in bins exposes glibc's memory management routines to a security vulnerability. If a malicious user is able to overflow a dynamically allocated block of memory, that user could overwrite the next contiguous chunk header in memory. When the overflown chunk is unallocated, and thus in a bin's double linked list, the attacker can control the values of that chunk's forward and back pointers. Given this information, consider the unlink macro used by glibc shown below:

```
     #define unlink(P, BK, FD) {   \
[1]     FD = P->fd;                \
[2]     BK = P->bk;                \
[3]     FD->bk = BK;               \
[4]     BK->fd = FD;               \
     }
```

Intended to remove a chunk from a bin's free list, the unlink routine can be subverted by a malicious user to write an arbitrary value to any address in memory.

In the unlink macro shown above, the first parameter P points to the chunk that is about to be removed from the double linked list. The attacker has to store the address of a pointer (minus 12 bytes, as explained below) in P→fd and the desired value in P→bk. At line [1] and [2], the values of the forward (P→fd) and back pointer (P→bk) are read and stored in the temporary variables FD and BK, respectively. At line [3], FD gets dereferenced and the address located at FD plus 12 bytes (the offset of the bk field within a boundary tag) is overwritten with the value stored in BK. This technique can be utilized, for example, to change an entry in the program's GOT (Global Offset Table) and redirect a function pointer to code of the attacker's choice.

A similar situation occurs with the frontlink macro (shown in Figure 1).

The task of this macro is to store the chunk of size S, pointed to by P, at the appropriate position in the double linked list of the bin with index IDX. FD is initialized with a pointer to the start of the list of the appropriate bin at line [1]. The loop at line [2] searches the double linked list to find the first chunk that is larger than P or the end of the list by following consecutive forward pointers (at line [3]). Note that every list stores chunks ordered by increasing sizes to facilitate a fast smallest-first search in case of memory allocations. When an attacker manages to overwrite the forward pointer of one of the traversed chunks with the address of a carefully crafted fake chunk, he could trick frontlink into leaving the loop (at line [2]) with FD pointing to this fake chunk. Next, the back pointer BK of that fake chunk would be read at line [4] and the integer located at BK plus 8 bytes (8 is the offset of the fd field within a boundary tag) would be overwritten with the address of the chunk P at line [5].

The attacker could store the address of a function pointer (minus 8 bytes) in the bk field of the fake chunk, and therefore trick frontlink into overwriting

```
#define frontlink(A, P, S, IDX, BK, FD) {          \
    ...
[1] FD = start_of_bin(IDX);
[2]     while ( FD != BK && S < chunksize(FD) ) {  \
[3]         FD = FD->fd;                           \
        }                                          \
[4]     BK = FD->bk;                               \
        ...
[5]     FD->bk = BK->fd = P;                       \
```

Figure 1: frontlink Macro.

this function pointer with the address of the chunk P at line [5]. Although this macro does not allow arbitrary values to be written, the attacker may be able to store valid machine code at the address of P. This code would then be executed the next time the function pointed to by the overwritten integer is called.

```
struct malloc_chunk
{
    INTERNAL_SIZE_T prev_size;
    INTERNAL_SIZE_T size;
    struct malloc_chunk *bk;
    struct malloc_chunk *fd;
};
```

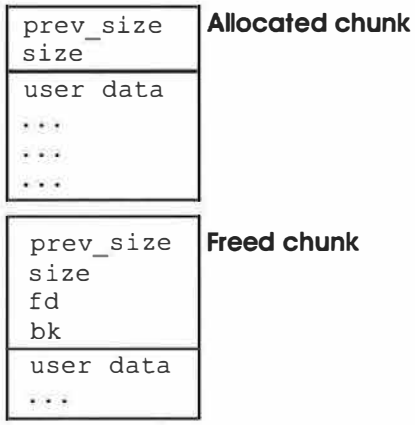


Figure 2: Original memory chunk structure and memory layout.

A variation on the heap overflow exploit described above is also possible, involving the manipulation of a chunk's size field instead of its list pointers. An attacker can supply arbitrary values to an adjacent chunk's size field, similar to the manipulation the list pointers. When the size field is accessed, for example during the coalescing of two unused chunks, the heap management routines can be tricked into considering an arbitrary location in memory, possibly under the attacker's control, as the next chunk. An attacker can set up a fake chunk header at this location in order to perform an attack as discussed above. If an attacker is, for some reason, unable to write to the list pointers of an adjacent chunk header but is able to reach the adjacent chunk's size field, this attack represents a viable alternative.

Heap Integrity Detection

In order to protect the heap, our system makes several modifications to glibc's heap manager, both in the structure of individual chunks as well as the management routines themselves.

```
struct malloc_chunk
{
  INTERNAL_SIZE_T magic;
  INTERNAL_SIZE_T __pad0;
  INTERNAL_SIZE_T prev_size;
  INTERNAL_SIZE_T size;
  struct malloc_chunk *bk;
  struct malloc_chunk *fd;
};
```

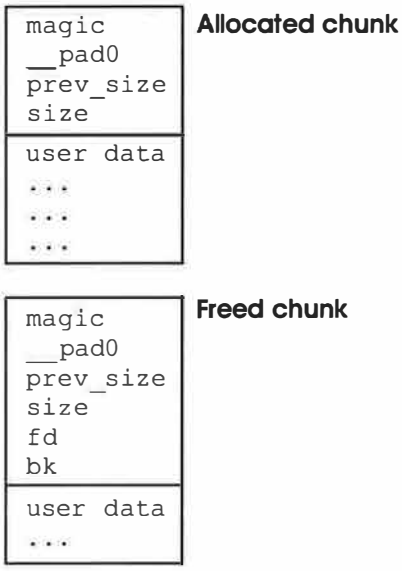


Figure 3: Modified memory chunk structure and memory layout.

Figure 2 depicts the original structure of a memory chunk in glibc.

The first element in protecting each chunk's management information is to prepend a canary to the chunk structure, as shown in Figure 3. An additional padding field, __pad0, is also added (dlmalloc requires the size of a header of a used chunk to be a power of two). The canary contains a checksum of the chunk header seeded with a random value, described below.

The second necessary element of our heap protection system is to introduce a global checksum seed value, which is held in a static variable (called __heap_magic). This variable is initialized during process startup with a random value, which is then protected against further writes by a call to mprotect. This is in contrast to stack protection schemes [29] that rely on repetitive calls to mprotect; since we only require a single invocation during process startup, we do not suffer from any related run-time performance loss associated with other schemes.

The final element of the heap protection system is to augment the heap management routines with code to manage and check each chunk's canary. Newly allocated chunks to be returned from malloc have their canary initialized to a checksum covering their memory location and size fields, seeded with the global value of __heap_magic. Note that the checksum function does not cover the list pointer fields for allocated chunks, since these fields are part of the chunk's user data section. The new chunk is then released to the application.

When a chunk is returned to the heap management through a call to free, the chunk's canary is checked against the checksum calculation performed when the chunk was released to the application. If the stored value does not match the current calculation, a corruption of the management information is assumed. At this point, an alert is raised, and the process is aborted. Otherwise, normal processing continues; the chunk is inserted into a bin and coalesced with bordering free chunks as necessary. Any free list manipulations which take place during this process are prefaced with a check of the involved chunks' canary values. After the deallocated chunk has been inserted into the free list, its canary is updated with a checksum covering its memory location, size fields, and list pointers, again seeded with the value of __heap_magic.

The elements described above effectively prevent writes to arbitrary locations in memory by modifying a chunk's header fields without being detected, whether through an overflow into or through direct manipulation of the chunk header fields. Each allocated chunk is protected by a randomly-seeded checksum over its memory location and size fields, and each free chunk is protected by a randomly-seeded checksum over its memory location, size fields, and list pointers. Each access of a list pointer is protected by a check to insure that the integrity of the pointers has not been violated. Also, each use of the size field is protected. Furthermore, the checksum seed has been protected against malicious writes to guarantee that it cannot be overwritten with a value chosen by the attacker.

As a beneficial side-effect, common programming errors such as unintended heap overflows or double invocations of free are detected by this system as well. A double call to free refers to the situation where a programmer mistakenly attempts to deallocate the same chunk twice. This error is detected due to a checksum mismatch. When the chunk is deallocated for the first time, its canary is updated to a new value reflecting its position on the free list. When the second call to free is executed, the checksum is checked again, with the assumption that it is an allocated chunk. However, since the canary has been updated and the check fails, an alarm is raised.

A limitation of our approach is the fact that we do not address general pointer corruption attacks, such as subversion of an application's function pointers. The system does not guarantee the integrity of user data contained within chunks in the heap; rather, the system guarantees only that the chunk headers themselves are valid.

It is also worth noting that the heap implementation included with glibc already contains functionality that attempts to ensure the integrity of the heap management information for debugging purposes. However, use of the debugging routines incurs significant cost in a production environment. The routines perform a full scan of the heap's free lists and global state during each execution of a heap management function, and include checks unrelated to heap pointer exploitation. Furthermore, there is no guarantee that all attacks are detected. Not all list manipulations are checked, and malicious values could pass integrity checks which are not specifically intended to protect against malicious overflows. Thus, we conclude that the included debugging functionality is not suitable for protecting against the vulnerabilities that we address.

The system described above has been implemented for glibc 2.3 and glibc 2.2.9x, pre-release versions of glibc 2.3 utilized by RedHat 8.0. However, the techniques developed for glibc are easily adaptable to other heap designs, including those shipped with the various BSD derivatives or commercial Unix implementations. Thus, further work is planned to apply this technique to other popular open systems besides glibc.

Evaluation

The purpose of this section is to experimentally verify the effectiveness of our heap protection technique. We also discuss the performance impact of our proposed extension and its stability.

To assess the ability of our protection scheme, we obtained several real-world exploits that perform heap overflow attacks against vulnerable programs. These were

- WU-Ftpd File Globbing Heap Corruption Vulnerability [30] against wuftpd 2.6.0,
- Sudo Password Prompt Heap Overflow Vulnerability [31] against sudo 1.6.3, and
- CVS Directory Request Double Free Heap Corruption Vulnerability [32] against cvs 1.11.4.

In addition, we used two proof-of-concept programs presented in [8] that demonstrate examples of

the exploit techniques using the unlink and the frontlink macro, respectively. We also developed a variant of the unlink exploit to demonstrate that dlmalloc's debugging routines can be easily evaded and do not provide protection comparable to our technique.

All vulnerable programs were run under RedHat Linux 8.0. The exploits have been executed three times, once with the default C library (i.e., glibc 2.2.93), once with the patched library including our heap integrity code, and once with the default C library and enabled debugging. The third run was performed to determine the effectiveness of the built-in debugging mechanisms in detecting heap-based overflows.

Table 1 shows the results of our experiments. A column entry of 'shell' indicates that an exploit was successful and provided an interactive shell with the credentials of the vulnerable process. A 'segfault' entry indicates that the exploit successfully corrupted the heap, but failed to run arbitrary code (note that it might still be possible to change the exploit to gain elevated privileges). 'aborted' means that the memory corruption has been successfully detected and the process has been terminated.

The results show that our technique was successful in detecting all corruptions of in-bound management information, and safely terminated the processes. Note that the built-in debugging support is also relatively effective in detecting inconsistencies, however, it does not offer complete protection and imposes a significantly higher performance penalty than our patch.

The performance impact of our scheme has been measured using several micro- and macro-benchmarks. We are aware of the fact that the memory management routines are an important part of almost all applications, and therefore, it is necessary to implement them efficiently. It is obvious that our protection approach inflicts a certain amount of overhead, but we also claim that this overhead is tolerable for most real-world applications and is easily compensated for by the increase in security.

To get a baseline for the worst slowdown that can be expected, we wrote a simple micro-benchmark that allocates and frees around four million (to be more precise, 2^{22}) objects of random sizes between 0 and 1024 bytes in a tight loop. The maximum size of 1024 was chosen to obtain a balanced distribution of objects in dedicated bins (for chunks with sizes less than 512 bytes) and objects in bins that cover a range of different sizes (for chunks with sizes greater than or equal to 512 bytes). We also utilized the dynamic memory benchmark present in the AIM 9 test suite [33]. Table 2 shows the average run-time in milliseconds over 100 iterations for the two micro-benchmarks. We provide results for a system with the default glibc, the glibc with heap protection and the glibc with debugging.

For more realistic measurements that reflect the impact on real-world applications, we utilized Mindcraft's WebStone [34] and OSDB [35]. WebStone is a client-server benchmark for HTTP servers that issues a number of HTTP GET requests for specific pages on a Web server and measures the throughput and response latency of each HTTP transfer. OSDB (open source database benchmark) is a benchmark that evaluates the I/O throughput and general processing power of GNU Linux systems. It is a test suite built on AS3AP, the ANSI SQL Scalable and Portable Benchmark, for evaluating the performance of database systems.

Figure 4 and Figure 5 show the throughput and the response latency measurements for an increasing number of HTTP clients in the WebStone benchmark, for both the default glibc and the patched version. We used an Intel Pentium 4 with 1.8 GHz, 1 GB RAM, Linux RedHat 8.0, and a 3COM 905C-TX NIC for the experiments, running Apache 2.0.40. It can be seen that even for hundred simultaneous clients, virtually no performance impact was recorded. Similar results have been obtained for OSDB 0.15.1. The following Table 3 shows the measurements for 10 parallel clients that used our test machine (the same as above) to full capacity, running a PostgreSQL 7.2.3 database. The results show the total run-time in seconds for the single-user and multi-user tests.

We also attempted to assess the stability of the patched library over an extended period in time. For this

Package	glibc	glibc + heap prot.	glibc + debugging
WU-Ftpd	shell	aborted	aborted
Sudo	shell	aborted	aborted
CVS	segfault	aborted	aborted
unlink	shell	aborted	aborted
frontlink	shell	aborted	aborted
debug evade	shell	aborted	shell

Table 1: Detection effectiveness.

Package	glibc	glibc + heap prot.	glibc + debugging
Loop	1,587	2,033 (+ 28%)	2,621 (+ 65%)
AIM 9	5,094	5,338 (+ 5%)	7,603 (+ 49%)

Table 2: Micro-Benchmarks.

purpose, the patch was installed on the Lab's web server (running Apache 2.0.40) and CVS server (running cvs 1.11.60). A patched library was also used on two desktop machines, running RedHat 8.0 and Gentoo 1.4, respectively. Although the web server only receives a small number of requests, the CVS server is regularly used for our software development and the desktop machines are the workstations of two of the authors. All machines were stable and have been running without any problems for a period of several weeks.

Package	glibc	glibc + heap prot.
OSDB	6,015	6,070 (+ 0.91%)

Table 3: OSDB benchmark.

Installation

Several methods of deploying our heap protection system have been developed, in order to accommodate various system environments and levels of desired protection. Many important security mechanisms are not applied because of the complexity and the required effort during setup. We provide different avenues that range from the installation of a pre-compiled package (with minimal effort) to a complete source rebuild of glibc.

One method is to download and install our library modifications as a source patch against glibc. Administrators can select the version appropriate to their system and apply it against a pristine glibc source tree before proceeding with the usual glibc source installation procedure. Source-based distributions, such as Gentoo Linux, can also easily incorporate these patches into their packaging system.

A second method of deploying is to create packages for various distributions of Linux that replace the system glibc image with a version containing our modifications (such as RedHat RPMs). The advantage of this approach is that virtually all applications on the target machine will be automatically protected against

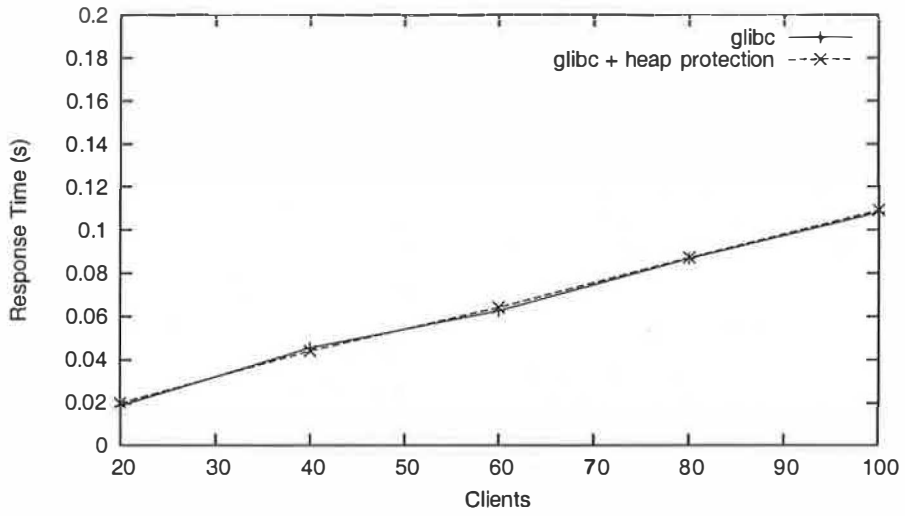


Figure 4: HTTP client response time.

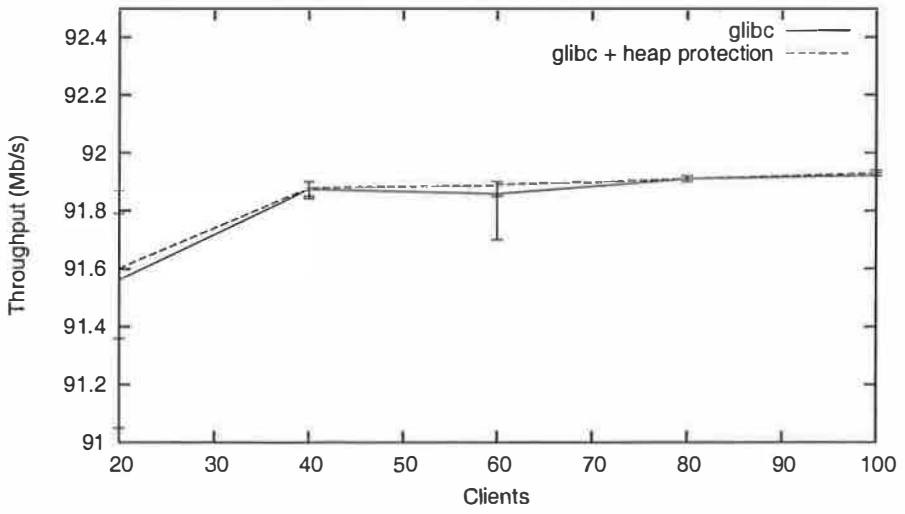


Figure 5: HTTP client throughput.

heap overflow exploitation, with the exception of those applications which are statically linked against glibc or perform their own memory management. A possible disadvantage is that these applications will also experience some level of performance degradation, which could be prohibitive in some high-performance environments.

A third method of deploying our heap protection system uses packages that install a protected glibc image alongside the existing image, instead of replacing the system's glibc image altogether. A script is provided that utilizes the system loader's LD_PRELOAD functionality to substitute the protected glibc image for the system image for an individual application. This allows an administrator to selectively enable protection only for certain applications (e.g., an administrator may not feel it necessary to protect applications which cannot be executed remotely, and therefore may wish to only protect those applications which are network-accessible). This is also a suitable path for admins that are afraid of potentially destabilizing their entire system by performing a system-wide deployment of a heap modification which has not undergone the extensive real-world testing that standalone dlmalloc has.

All of the described installation methods are documented in detail on our website, located at http://www.cs.ucsb.edu/~rsg/heap/. Packages for various popular distributions and source patches can be downloaded as well.

Conclusions

This paper presents a technique for detecting heap-based overflows that tamper with in-band memory management data structures. We discuss different ways to mount such attacks and show our mechanism to detect and prevent them. We implemented a patch for glibc 2.3 that extends the utilized data structures with a canary that stores a checksum over the sensitive data. This checksum calculation involves a secret seed that makes it infeasible for an intruder to guess or fake the canary in an attack.

Experience shows that system administrators are often reluctant to adopt security measures in the systems they administer. Installing new tools may require significant effort to understand how to best apply the technology in the administrator's network, as well as investment in training end users. Additionally, applying a new tool may interfere with existing critical systems or impose unacceptable run-time overhead. This paper introduces a heap protection mechanism that increases application security in a way that is nearly transparent to the functioning of applications and is invisible to users. Applying the system to existing installations has few drawbacks. Recompilation of applications is rarely required, and the system imposes minimal overhead on application performance.

Acknowledgments

This research was supported by the Army Research Office, under agreement DAAD19-01-1-0484. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon.

The views and conclusions contained herein are those of the author and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Army Research Office, or the U.S. Government.

Author Information

Christopher Kruegel is working as a research postgraduate in the Reliable Software Group at the University of California, Santa Barbara. Previously, he was an assistant professor at the Distributed Systems Group at the Technical University Vienna. Kruegel holds the M.S. and Ph.D. degrees in computer science from the Technical University Vienna. His research focus is on network security, with an emphasis on intrusion detection. You can contact him at chris@cs.ucsb.edu.

Darren Mutz is a doctoral student in the Computer Science department at the University of California, Santa Barbara. His research interests are in network security and intrusion detection. From 1997 to 2001 he was employed as a member of technical staff in the Planning and Scheduling Group at the Jet Propulsion Laboratory where he engaged in research efforts focused on applying AI, machine learning, and optimization methodologies to problems in space exploration. He holds a B.S. degree in Computer Science from UCSB and can be contacted at dhm@cs.ucsb.edu.

William Robertson is a first-year PhD student in the Computer Science department at the University of California, Santa Barbara. His research interests include intrusion detection, hardening of computer systems, and routing security. He received his B.S. degree in Computer Science from UC Santa Barbara, and can be reached electronically at wkr@cs.ucsb.edu.

Fredrik Valeur is currently a Ph.D. student at UC Santa Barbara. He holds a Sivilingenioer degree in Computer Science from the Norwegian University of Science and Technology. His research interests include intrusion detection, network security and network scanning techniques. He can be contacted at fredrik@cs.ucsb.edu.

References

[1] Spafford, E., "The Internet Work Program," *Analysis Computer Communication Review*, 1998.

[2] AlephOne, *Smashing the Stack for Fun and Profit*, http://www.phrack.org/phrack/49/P49-14.

[3] Wilander, J. and M. Kamkar, "Comparison of Publicly Available Tools for Dynamic Buffer Overflow Prevention," *10th Network and Distributed System Security Symposium*, 2003.

[4] *CERT Advisory CA-2002-11*, "Heap Overflow in Cachefs Daemon (cachefsd)," http://www.cert.org/advisories/CA-2002-11.html.

[5] *CERT Advisory CA-2002-33*, "Heap Overflow Vulnerability in Microsoft Data Access Components (MDAC)," http://www.cert.org/advisories/CA-2002-33.html.

[6] Conover, M., *w00w00 on Heap Overflows*, http://www.w00w00.org/files/articles/heaptut.txt.

[7] anonymous, *Once upon a free()*, http://www.phrack.org/phrack/57/p57-0x09.

[8] Kaempf, M., *Vudo malloc tricks*, http://www.phrack.org/phrack/57/p57-0x08.

[9] Designer, Solar, *JPEG COM Marker Processing Vulnerability in Netscape Browsers*, http://www.openwall.com/advisories/OW-002-netscape-jpeg.txt.

[10] *The GNU C Library*, http://www.gnu.org/software/libc/libc.html.

[11] Cowan, C., et al., "StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks," *7th USENIX Security Conference*, 1998.

[12] Vendicator, *Stack Shield Technical Info*, http://www.angelfire.com/sk/stackshield/index.html.

[13] Baratloo, A., N. Singh and T. Tsai, *Libsafe: Protecting critical elements of stacks*, http://www.research.avayalabs.com/project/libsafe/index.html.

[14] Baratloo, A., N. Singh and T. Tsai, "Transparent Run-time Defense Against Stack Smashing Attacks," *USENIX Annual Technical Conference*, 2000.

[15] Designer, Solar, *Non-executable stack patch*, http://www.openwall.com.

[16] *RSX: Run-time addressSpace eXtender*, http://www.starzetz.com/software/rsx/index.html.

[17] *PAX: Non-executable heap-segments*, http://pageexec.virtualave.net/index.html.

[18] *Valgrind, an open-source memory debugger for x86-GNU/Linux*, http://developer.kde.org/~sewardj/index.html.

[19] *Electric Fence – Memory Debugger*, http://www.gnu.org/directory/devel/debug/ElectricFence.html.

[20] Huang, Y., *Protection Against Exploitation of Stack and Heap Overflows*, http://members.rogers.com/exurity/pdf/AntiOverflows.pdf.

[21] Necula, George C., Scott McPeak, and Westley Weimer, "CCured: Type-safe retrotting of legacy code," *29th ACM Symposium on Principles of Programming Languages*, 2002.

[22] Dean, D., E. Felten and D. Wallach, "Java Security: From HotJava to Netscape and Beyond," *IEEE Symposium on Security and Privacy*, 1996.

[23] *The Last Stage of Delirium (LSD), Java and Java Virtual Machine Vulnerabilities and their Exploitation Techniques*, http://www.lsd-pl.net/java_security.html.

[24] *ISO JTC 1/SC 22/WG 14 – C*, http://std.dkuug.dk/JTC1/SC22/WG14/index.html.

[25] *ISO JTC 1/SC 22/WG 15 – POSIX*, http://std.dkuug.dk/JTC1/SC22/WG15/index.html.

[26] *The GNU C Library Manual*, http://www.gnu.org/manual/glibc-2.2.5/libc.html.

[27] Lea, D., *A Memory Allocator*, http://gee.cs.oswego.edu/dl/html/malloc.html.

[28] Wilson, P., M. Johnstone, M. Neely, and D. Boles, "Dynamic Storage Allocation: A Survey and Critical Review," *International Workshop on Memory Management*, 1995.

[29] Chiueh, T., and F. Hsu, "RAD: A Compile-time Solution to Buffer Overflow Attacks," *21st Conference on Distributed Computing Systems*, 2001.

[30] *WU-Ftpd File Globbing Heap Corruption Vulnerability*, http://www.securityfocus.com/bid/3581.

[31] *Sudo Password Prompt Heap Overflow Vulnerability*, http://www.securityfocus.com/bid/4593.

[32] *CVS Directory Request Double Free Heap Corruption Vulnerability*, http://www.securityfocus.com/bid/6650.

[33] *AIM IX Benchmarks*, http://www.caldera.com/developers/community/contrib/aim.html.

[34] *Mindcraft WebStone – The Benchmark for Web Servers*, http://www.mindcraft.com/benchmarks/webstone/.

[35] *OSDB – The Open Source Database Benchmark*, http://osdb.sourceforge.net.

Designing a Configuration Monitoring and Reporting Environment

Xev Gittler and Ken Beer – Deutsche Bank

ABSTRACT

The Configuration Monitoring and Reporting Environment (CMRE) is a tool designed to collect and report on the many configuration details of systems within an enterprise. It is designed to gather information on systems about which initially very little is known. CMRE needs few prerequisites in order to do its job. CMRE is modular, flexible and runs on many different platforms. It is written in a combination of Perl, Korn shell and PHP and uses proprietary as well as open-source software. CMRE currently collects data on thousands of UNIX and Windows systems at Deutsche Bank world wide. This paper will describe the conditions that led to the need for such a tool, its design and limitations as well as the difficulties found along the way. This paper will also touch upon our performance monitoring tool (PMRE), whose data we use to help us understand our systems environment.

Accommodating Rapid Growth

Over the last ten years, Deutsche Bank (DB) has executed a strategy which has led to its rapid growth as a dominant player in the investment banking industry. This strategy involved organic growth, acquisitions and mergers. The natural outgrowth of this strategy is that our systems environment contains a diverse set of systems built in a number of organizations, with differing standards and procedures. There was an obvious case for improving the infrastructure by consolidating our infrastructure to provide an environment that is more efficient, uniform and easy to service. We wanted to provide a platform that would allow us to more easily answer questions from management about configuration and security issues.

Some of the questions that we wanted to be able to more easily answer include:

- Security: List all machines that are patched appropriately
- Capacity Planning: List number of machines with empty CPU slots.
- Physical moves: List all machines pointing to a particular DNS server

Given that the different environments came with their own standards and procedures, we wanted to develop a uniform mechanism for retrieving information to enable us to understand our environment as a whole. Once we had a uniform method, we could further improve our environment by adding proactive management tools.

Our goal is to proactively manage all our systems, including all configuration options. We want to be able to specify from a central location exactly how a system is configured without having to wait for an SA to specify the data, but which requires some initial consistency, which we cannot easily get until we have a good view of what the systems look like.

Design Specifications

When designing the CMRE system, our primary goals were to put a system in place that could be deployed quickly (within a few months), did not require significant effort or expense, yet vastly improved our ability to understand our environment. We examined a number of systems deployed internally as well as external products, such as Explorer, cfengine, and SNMP-based solutions, but ultimately determined that the simplest, most flexible method was to develop a single homegrown system, based on the design principles of an existing tool already deployed in one of our branch offices.

We needed to develop an inexpensive, simple, powerful, multiplatform solution. The client software would run as root on every UNIX system at DB globally, but would switch to the minimal permissions required for each particular command. Therefore, we needed to ensure that the software would be safe and would not add unexpected load to our client systems. We could not risk using software that we did not completely understand. On the other hand, we could not build a lab to fully test the software because we did not know what kinds of systems were installed. Finding, testing, and adapting another's solution was therefore deemed to be as time consuming as writing a simple custom application.

In order to determine the primary functionality of the system, we conducted extensive discussions with the various groups that we considered stakeholders in this project and took their needs into account:

- System Administrators – need to examine the systems in detail, and view consolidated system data in order to troubleshoot, upgrade and move systems.
- Security/Audit – need to examine summary reports and drill down to see specific problems.

- Developers – need to analyze the impact of their applications on their systems and determine resources available to them.
- Engineering groups – need to find systemic bottlenecks and need to be able to plan for infrastructure changes.
- Business Managers – need to examine resource utilization and system status across their business.
- Senior Management – need to see high level overviews of status and performance across all systems globally, summarized by group.

Based on the above, we determined that our configuration monitoring system should provide:

- **Accessible configuration information** – The basic feature of the system should be to take data that was previously only available in many different, scattered, often inaccessible locations and gather and present it from a single location.
- **Planning data** – We required a comprehensive and easily accessible company-wide picture of systems in order to plan for future growth. For instance, without knowing how many machines were running Solaris 2.6, we could not assess the size of the task of upgrading these systems when the operating system was at end-of-life. Nor could we determine when applications could be retired if we did not know the application use metrics.
- **Problem Detection and Repair** – We required the ability to identify and correct system specific problems in all affected machines as soon as the problem arose or even before it became an issue. For instance, when we found an issue, we wanted to check all systems that might be similarly affected and either patch the system or mark it so that when the problem occurred, we would be able to quickly determine the corrective course of action. Similarly, we needed to be able to identify and repair security and audit issues. Without system configuration information, we had to manually examine each and every system to determine whether security or audit issues affected the system.
- **Historical Configuration Information** – We wanted to gather and store a significant amount of configuration data about each system in our environment, including historical information for comparison purposes.
- **Flexibility** – We wanted to add additional information collectors and reports easily as new requirements arose.
- **Support for Multiple Operating Systems** – We wanted the system to work across all our supported operating systems, gathering detailed information on all systems.
- **Audience Specific Views** – We wanted to be able to present support group level summarizations of configuration data, such as a list of all systems within a group running a particular operating system, or running on particular hardware, and view summary information via a web front end. In addition, we wanted to present management with global summarizations of configuration data
- **Data Filtering** – We wanted to filter data, such as flagging all systems that did not meet a particular standard, such as a minimal operating system level, or a particular security patch.
- **External Data Connectivity** – We wanted to tie into our inventory, monitoring, security and other databases, and help us improve the quality of that data.

The overarching design goal was to create a single portal where users could find any and all configuration information they might require, regardless of whether we changed the underlying tool used to collect the information. We wanted a single place that provided all the configuration information that one might require.

Recognizing that information gathering and reporting requirements would grow in sophistication as people began to use these systems, the system had to be designed with the flexibility to grow in unexpected ways. This required the ability to tailor reports to various audiences (e.g., CIOs, IT Senior Management, applications/development managers, support managers, and system and database administrators).

The Challenges

Once we had a basic design concept we needed to make it work within the diverse environment that currently existed. This section will discuss the various issues that we ran across and how we addressed them. Many of these challenges are a direct result of combining organizations and systems as part of our company's rapid growth.

Multiple Operating Systems and Revisions

As is typical in investment banks, we are required to support a number of different operating systems and versions supporting different businesses. We needed to support a range of operating systems, including multiple versions of Solaris, AIX, Linux, HPUX and Windows. While we were willing to support different sets of scripts for Windows and Unix, we wanted to maintain a single set of scripts for all Unix platforms. We also wanted output to be as similar as possible from all operating systems.

No Guarantee Of Tools On Client Systems

Because we had grown from multiple environments, one of the first major hurdles we came across was that there was not consistent tool set available across all systems. Our initial thought was to write the tool in Perl, but we were unable to do so because most of our machines did not have Perl installed. We considered deploying Perl with the client software, but decided coordinating multiple versions of Perl would

be more trouble than it was worth. We chose Korn shell (ksh) as a common denominator and did not use any external tools that did not install by default on all operating systems we were using.

File System Layout and Disk Space Allowance

Since our systems were installed by a wide variety of groups that were working from different specifications, we could not rely on any particular file system to be available. Some machines had /opt, some had /apps, some /usr/local. In some cases, /usr/local was NFS mounted read-only. Many of the systems had minimal disk configurations, so we had to ensure that both our scripts and their output fit in 10-30 MB of space.

Minimal Client Processing

Many of our business areas are extremely sensitive to any additional load placed on their systems. Because of this restriction we had to minimize the processing of data on the client system. Our experience is that should our software interrupt a running application, the application's business owner could forbid the running of our code on their systems and force us to use a custom solution (or do without).

Different Trust Mechanism

Throughout the bank we had many different mechanisms for allowing access to hosts, including SSH based golden hosts[1] for large groups and a variety of other mechanisms for smaller groups. For CMRE, however, we would require access to all systems across the bank.

Unexpected Configurations

When writing our scripts, we tested across a reasonable variety of systems. However interesting things can happen when, for instance, we run a disk

[1] A golden host is a well secured machine that has secure access to other machines. It is used for a variety of system administration purposes.

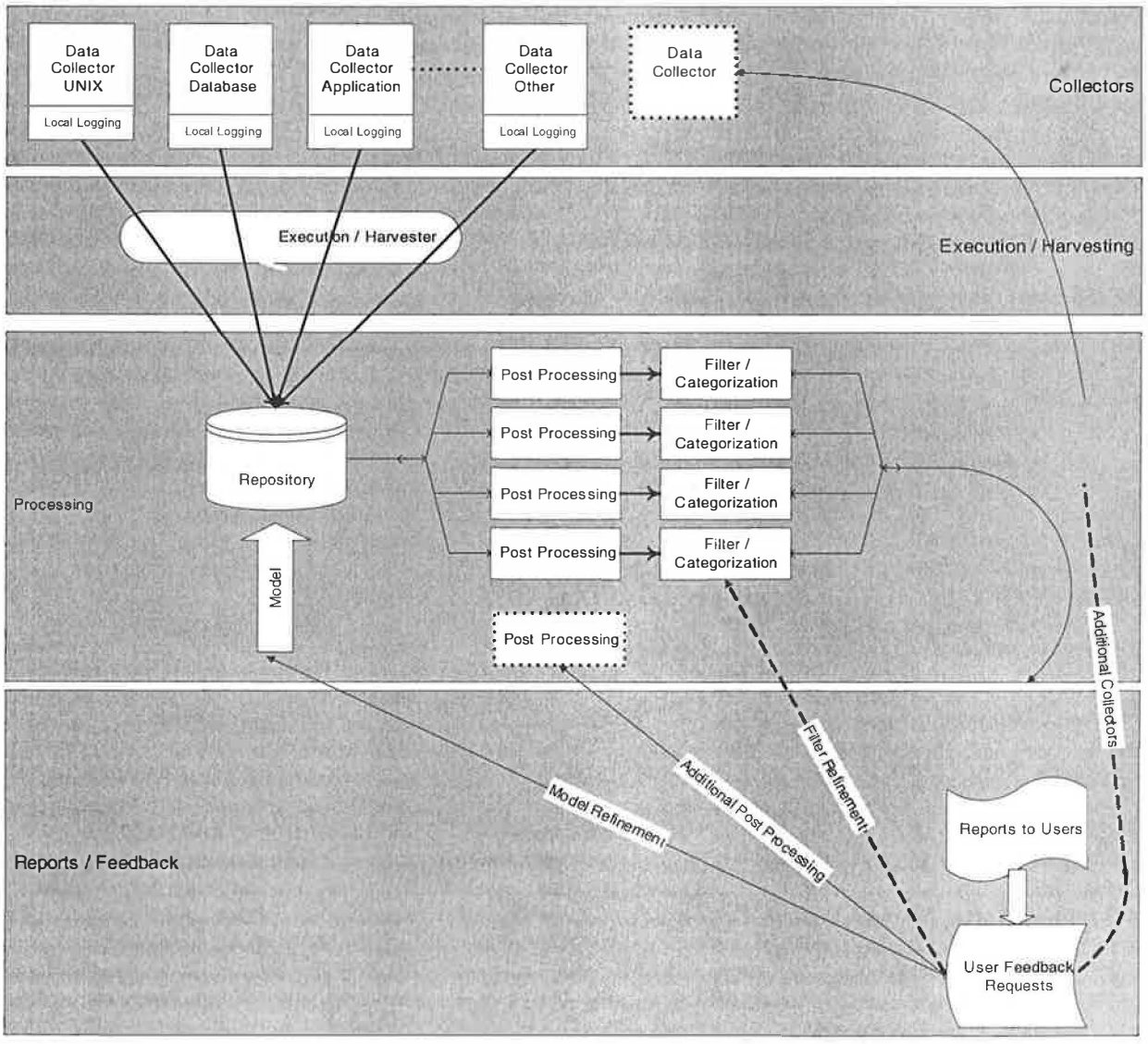


Figure 1: Block diagram of CMRE.

information command on a system with over 3,000 attached disks. As we find edge-cases like this, we modified our scripts to be even more careful and generic. Often the unexpected actions happen because of an incorrect system configuration. The impulse is to say, "You should not put swap on a CDRW" or "you should not run 10 copies of Oracle on an Ultra1," however there may be perfectly valid reasons for these configurations. It cannot be a requirement of CMRE for a system to be configured in a particular way. CMRE's job is to collect data from every system, regardless of how oddly configured. Improving the infrastructure is a separate task.

The Tool

The tool that we designed is part of a framework we call the Monitoring and Reporting Environment (MRE). This paper discusses only the Configuration (CMRE) and (briefly) the Performance (PMRE) components.

The CMRE architecture is based on a simple, modular framework. CMRE consists of

Category	Sys Type	Sys Vers	Cmd/ File	Run as	Output Filename	Command or File	Description
Automount	All	All	C	Nobody	lsautomnt	ls -al /etc/auto_*	Get timestamp of automounter files
Automount	All	All	F	Nobody		/etc/auto_*	Get automounter files
Config	SunOS	All	C	Nobody	lsgrpsys	ls -al /etc/system	Get kernel config
Config	All	All	C	Nobody	lsvarspl	ls -laR /var/spool/cron	Get timestamps on cron directory tree
Config	All	All	C	Nobody	hostid	hostid	Get the unix host id
Config	All	All	C	Nobody	uptime	uptime	Get the time alive
Config	All	All	C	Nobody	date	date -u	Date Stamp
Config	All	All	C	Nobody	findperl	findperl.sh	Find where perl is
Config	All	All	F	Nobody		/etc/profile	Get the host default user profile
Config	All	All	F	Nobody		/etc/services	Get the local services file
Config	All	All	F	Nobody		/etc/protocols	Get the local protocols file
Config	All	All	F	Nobody		/opt/mitk5/etc/krb5.conf	MIT K5 config file
Config	All	All	F	Nobody		/opt/mitk5/etc/ad.conf	MIT K5 config file
Config	All	All	F	Root		/.login	root's login info
Config	All	All	F	Root		/.cshrc	root's login info
Config	SunOS	All	C	Nobody	lpstat	lpstat -t	Get printer config
Config	SunOS	All	C	Root	orcaperf	orcaperf.sh	Get the orca performance data
Config	SunOS	All	C	Nobody	dmesg	dmesg	SunOS specific command to get kernel messages
Config	SunOS	All	C	Nobody	uname	uname -a	Get the unix machine/OS level id info
Config	SunOS	All	C	Nobody	swap-s	swap -s	Get the active swap space
Config	SunOS	All	C	Nobody	swap-l	swap -l	List the swap areas
Config	SunOS	All	C	Nobody	pkginfo	pkginfo -l	Dump the verbose package listing
Config	SunOS	All	C	Nobody	modinfo	modinfo	Dump the kernel modules
Config	SunOS	All	C	Nobody	showrev	showrev -a	Show package and patch info
Filesystem	SunOS	All	C	Nobody	dfufs	df -kF ufs	Display all ufs filesystems
Filesystem	SunOS	All	C	Nobody	dfvxfs	df -kF vxfs	Display all veritas filesystems
Filesystem	SunOS	All	C	Nobody	dfnfs	df -kF nfs	Display all nfs mounts
Filesystem	SunOS	All	C	Root	veritas	veritas.sh	Display veritas info
Filesystem	SunOS	All	C	Root	veritas-ha	ha.sh	Display HA info
Filesystem	AIX	All	C	Nobody	mount	mount	Show what filesystems currently mounted
Filesystem	AIX	All	C	Nobody	lsfs	lsfs -c	AIX specific command to list filesystems
Filesystem	SunOS	All	F	Nobody		/etc/vfstab	SunOS specific file containing filesystem mount points
Filesystem	SunOS	All	F	Nobody		/etc/nfssec.conf	SunOS specific file for NFS security
Filesystem	AIX	All	F	Nobody		/etc/filesystems	AIX specific for filesystems
Filesystem	Linux	All	C	Nobody	dfext2-linux	df -kF ext2	Display all ext2 filesystems
Filesystem	Linux	All	C	Nobody	dfrfs-linux	df -kF reiserfs	Display all nfs mounts
Hardware	SunOS	All	C	Nobody	prtdiag	prtdiag -v	SunOS specific command displaying hardware info
Hardware	SunOS	All	C	Nobody	eeprom	eeprom -v	SunOS specific command displaying PROM settings

Table 1: Sample lines from master configuration file.

- Minimal collector software
- An execution component for running the collector software
- A harvesting component for gathering the data to a central location
- A processing component for aggregating and mining the data
- A browser-based reporting component

Figure 1 depicts the components of CMRE.

Collector

The collector is a set of Korn shell scripts and configuration files. Master.sh is the script that is run using either our installed framework product or our golden host infrastructure. It parses the appropriate commands file (described below) and uses it to acquire the appropriate files, and run the appropriate commands and scripts. Before they're run the commands and scripts are first wrapped in an OS-specific shell wrapper. This way collectors can be as simple as a single command yet still have appropriate variables set, traps, etc. STDOUT and STDERR are collected as well as measurements of the time the command took to complete and timeout alarms. Success or failure is recorded as well.

/.cshrc	/etc/inetd.conf	/etc/protocols	/etc/sudoers
/.login	/etc/init.d/*	/etc/rc*	/etc/syslog.conf
/etc/*.conf	/etc/inittab	/etc/resolv.conf	/etc/system
/etc/adsm	/etc/irs.conf	/etc/rpc	/etc/vfstab
/etc/auto_*	/etc/lvm	/etc/security/audit/config	/opt/mitk5/etc/ad.conf
/etc/cron.d/cron.allow	/etc/mail	/etc/security/audit/events	/opt/mitk5/etc/krb5.conf
/etc/cron.d/cron.deny	/etc/name_to_major	/etc/security/audit/objects	/var/adm/cron/cron.allow
/etc/default/*	/etc/named.conf	/etc/security/audit_control	/var/adm/cron/cron.deny
/etc/default/login	/etc/netgroup	/etc/security/audit_event	/var/adm/db/*
/etc/defaultdomain	/etc/netmasks	/etc/security/audit_user	/var/adm/loginlog
/etc/defaultrouter	/etc/netsvc.conf	/etc/security/failedlogin	/var/adm/messages
/etc/dfs/*	/etc/nfssec.conf	/etc/security/lastlog	/var/adm/sudo/sulog
/etc/exports	/etc/nodename	/etc/security/login.cfg	/var/adm/sulog
/etc/filesystems	/etc/nsswitch.conf	/etc/security/passwd	/var/adm/syslog.conf
/etc/group	/etc/oratab	/etc/security/user	/var/log/messages
/etc/hostname.*	/etc/passwd	/etc/sendmail.cf	/var/opt/oracle/oratab
/etc/hosts.equiv	/etc/path_to_inst	/etc/services	/var/spool/cron/allow
/etc/hosts	/etc/printers.conf	/etc/shadow	/var/spool/cron/deny
/etc/inet/*	/etc/profile	/etc/ssh/sshd_config	/var/yp/Makefile

Table 2: Files collected.

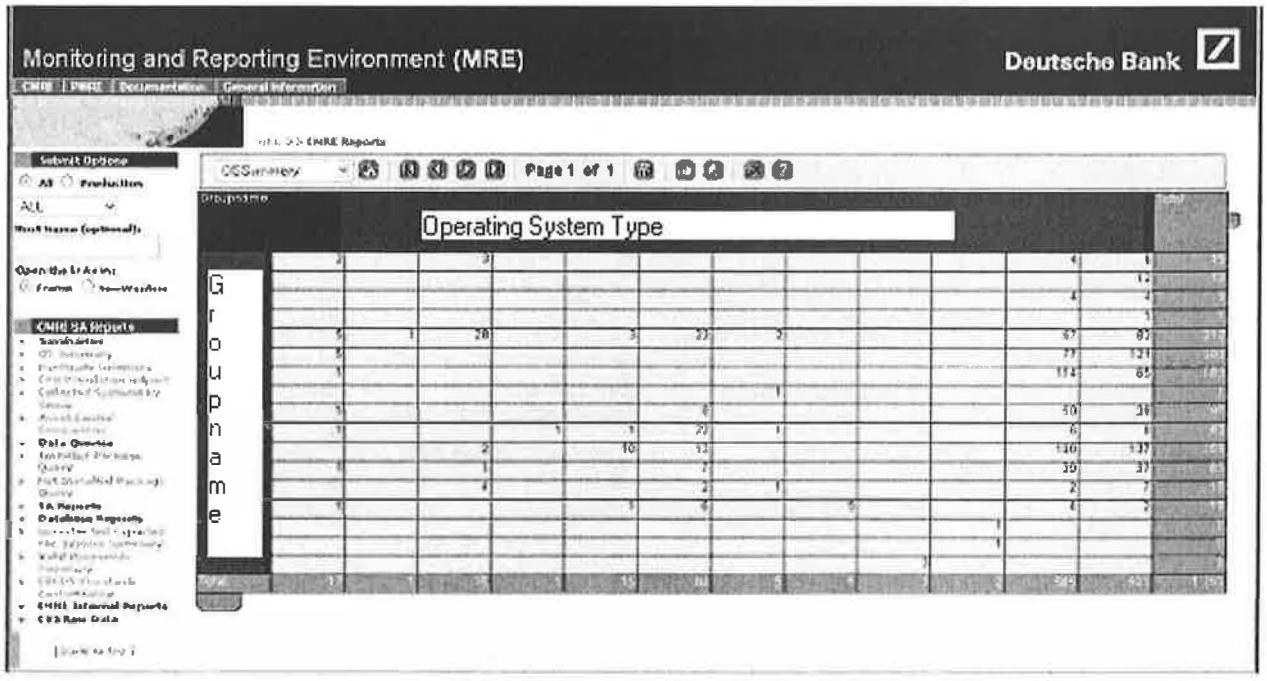


Figure 2: Sample OS count report.

There is a single master configuration file that is maintained centrally. An extract of the master configuration file is presented in the next section. The file is delimiter separated that determines which files get collected and which commands or custom scripts are run on what version(s) of Unix (and as what user). The collectors are grouped into multiple classes such as "Network" and "Config", as well as operating system specific classifications. This configuration file is broken out into operating system specific files which are packaged into the distribution of the software. Master.sh reads the specific configuration file to determine what commands to execute.

Note that Master.sh does not do much parsing of output. The output of many system commands is designed to be human-, not machine-readable so complex parsing is often required. Instead, parsing is done at the back end. CMRE is effectively a distributed system, yet we intentionally do not take advantage of it for parsing. While we do have concerns about adding load, breaking things, and leaving data behind, the primary reason not to parse the data on the front end has to do with the difference in deployment speed between the parts of CMRE. Because of testing, change control, and approval processes, new versions of the client software can take weeks to roll out. Back end parsing, however, can be changed on the fly. This makes it more effective for us to acquire as much of the raw data as possible and parse it later. When we discover that we now require additional information, we can immediately change our parsers, rather than wait for the next rollout.

Master.sh keeps two copies of the data it collects on the client system – today's data and yesterday's data. After collection, master.sh compares what is new or different from yesterday's run and puts just those files and output in a compressed tar file to be collected later.

It takes a certain amount of finesse to write the collector software. Care must be taken to write code that is multiplatform, stable, and with low system

/bin/echo dummy	ifconfig -a	lsdev -C	psrinfo -v
/bin/last -100	ipcs -a	lsdev -Cc processor	Ptree
/usr/ucb/ps -auxwwe	Logins	lsfs -c	Pwstats.sh
arp -a	lpstat -t	lslpp -h all	Rpcinfo -p 127.0.0.1
crontab -l	ls -al /etc /etc/*.conf /etc/passwd /etc/shadow	lslpp -L all	Sacadm -L
date -u	ls -al /etc/auto.*	lsps -a	Secpasswd.sh
df -kF ext2	ls -al /etc/auto_*	lspvaix.sh	Showmount -e localhost
df -kF nfs	ls -al /etc/defaultrouter	lssrc -a	Showrev -a
df -kF reiserfs	ls -al /etc/group	lsvgaix.sh	Sundisks.sh
df -kF ufs	ls -al /etc/mail/*	modinfo	Swap -l
df -kF vxfs	ls -al /etc/named.conf	mount	Swap -s
diskformat.sh	ls -al /etc/netmasks	netstat -a	Sybase.sh
dmesg	ls -al /etc/networks	netstat -rn	Sysdef -d | egrep -v '(driver.not.attached|no.driver)'
domainname	ls -al /etc/printers.conf	network.sh	Sysdef -I
dsmc q files	ls -al /etc/rpc	odmget -q attribute=realmem CuAt	Sysinfo -level all -format report
dsmc q sched	ls -al /etc/sendmail.cf	odmget -q name=interface CuAt	Titancheck.sh
dumpadm	ls -al /etc/system	oracle-adds.sh	Uname -a
eeprom -v	ls -alR /etc	orcaperf.sh	Uname -aM
errpt	ls -alR /var/spool/lp /var/spool/print	oslevel	Uptime
fbconfig -list	ls -altr /var/yp/domainname	pkginfo -l	Veritas.sh
filepermlist.sh	ls -l /dev/rdsk	pmadm -L	who -a
findperl.sh	ls -l /etc/defaultdomain	prtconf -vD	Ypcat -k rpc.bynumber
getaixos.sh	ls -l /etc/inittab	prtconf | awk -F: '/Memory/ {print $2}' | head -1	Ypcat -k ypservers
getrhosts.sh	ls -l /etc/resolv.conf	prtdiag -v	Ypmaps.sh
getsshkeys.sh	ls -l /tftpboot	ad l prtdiag10k -v | grep Memory Size | awk '{print $3}'	Ypwhich
ha.sh	ls -laR /var/spool/cron	ps -efal	Ypwhich -m
hostid	Lscfg	ps auwwxe	

Table 3: List of executed commands.

overhead. As most of the scripts are run as root across all machines in the company, security must also be considered very carefully.

Sample Collector Configuration

Table 1 shows some sample lines taken from the master configuration file while Table 2 shows the names of the all the files collected. Table 3 shows the commands that currently executed across the various operating systems.

Execution / Harvesting

Execution and harvesting proved very challenging initially. We came up with two methods of performing these tasks, described below. Each technique executes the collector script, which creates a single file on the client, 'config.tar.Z'. The harvester then collects the file from each machine and renames it to a unique name. It then tars all of the individual files into a single tar file and uploads that file to the central processing machine via SSH.

Management Framework

Deutsche Bank uses a commercial system management framework product on a majority of our machines. In fact, it is a requirement that all systems in datacenters run the framework's monitoring agents. The framework gives us a secure method of distributing software, executing commands on remote machines, and pulling the data back to a central location. Where the framework is installed, we use it to install the software (under a common directory structure provided by the

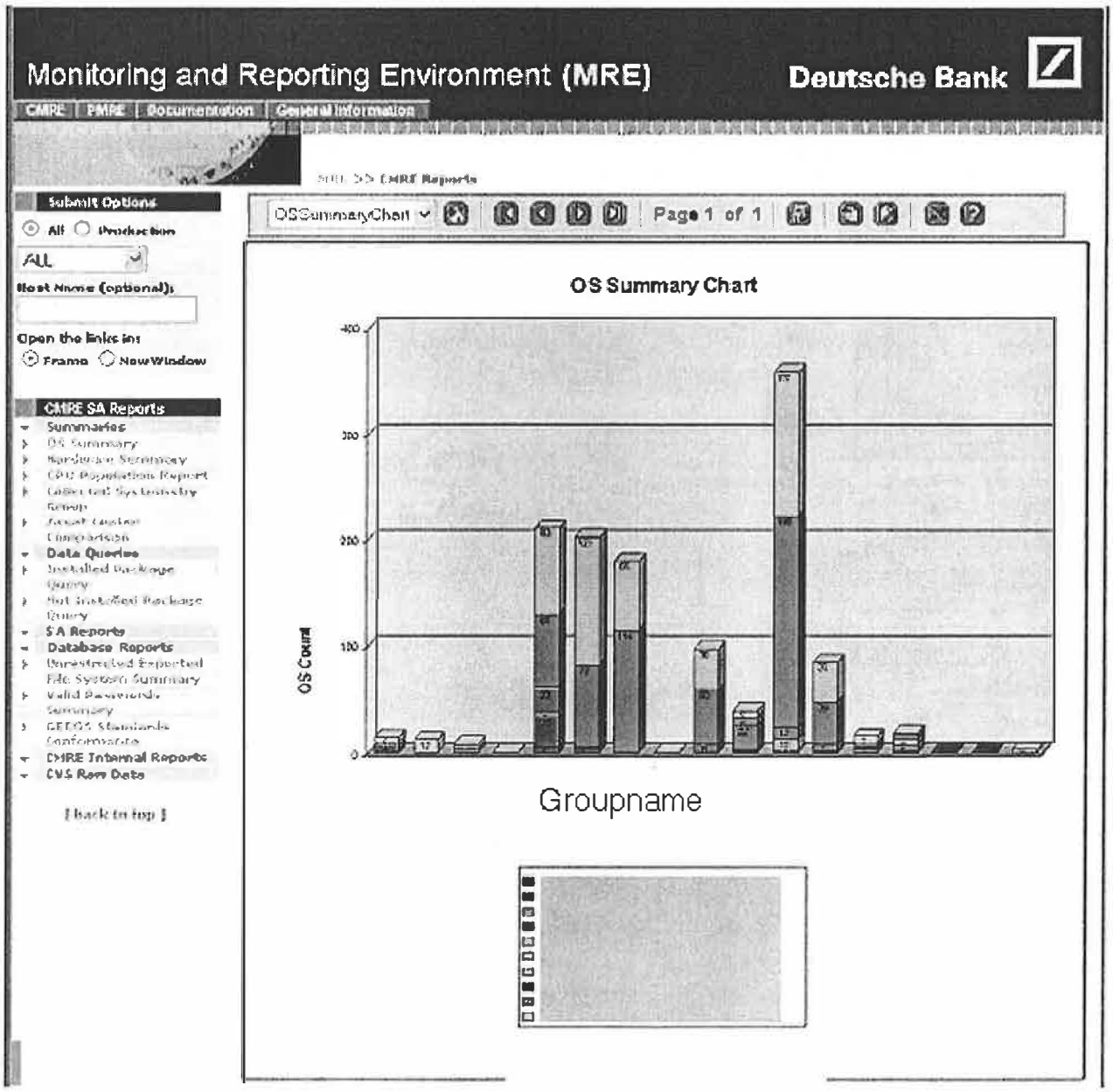


Figure 3: OS summary chart.

framework, so that we have a directory structure that we can count on), execute the collector on a nightly basis, and securely retrieve the information to the framework management nodes. From the framework management nodes we use SSH to get the aggregated data securely to our central CMRE processing server.

Golden Hosts

For those machines where the framework is not available, we have packaged the software including the appropriate crontab entries to execute the commands on a daily basis. We have also provided templates and instructions for the SA groups to assist them in setting up a harvester through whatever golden host mechanism that they employ, using SSH. We then use SSH to get the aggregated data securely to our central CMRE processing server. In this case, the System Administrator is required to ensure that there is a trust relationship, that the file systems that we require exist, and that enough space for collection is available.

Processing

After receiving the configuration bundles from the harvesting systems, we extract the contents into directories – one for each host. If the collector data we receive is a full collection, marked with a different filename by the collector, we first remove all the data in that host directory, and then we un-tar the file into the host directory. If it is an incremental collection, then we un-tar the file into the host directory, over-writing any changed files. After the files are extracted, we use CVS to check the files in, thereby keeping a historical record of the data.

We collect between 0.5 and 30 MB of raw data from each client (depending on the type and contents of each system). After the data is uploaded to our central server, we process some of that data and upload it into a database. This processing allows us to easily query and report on information, as well as aggregate data into group and company wide reports. The following is a sample of the data that we upload to the database:

- Date – one of the collectors executed is the 'date' command which we upload to determine the last time that the collection was successful.
- Operating System
- OS Version
- Hardware
- System Build Version – our internal identifier for a particular OS release that we make available
- Memory
- Number and Speed of Processors

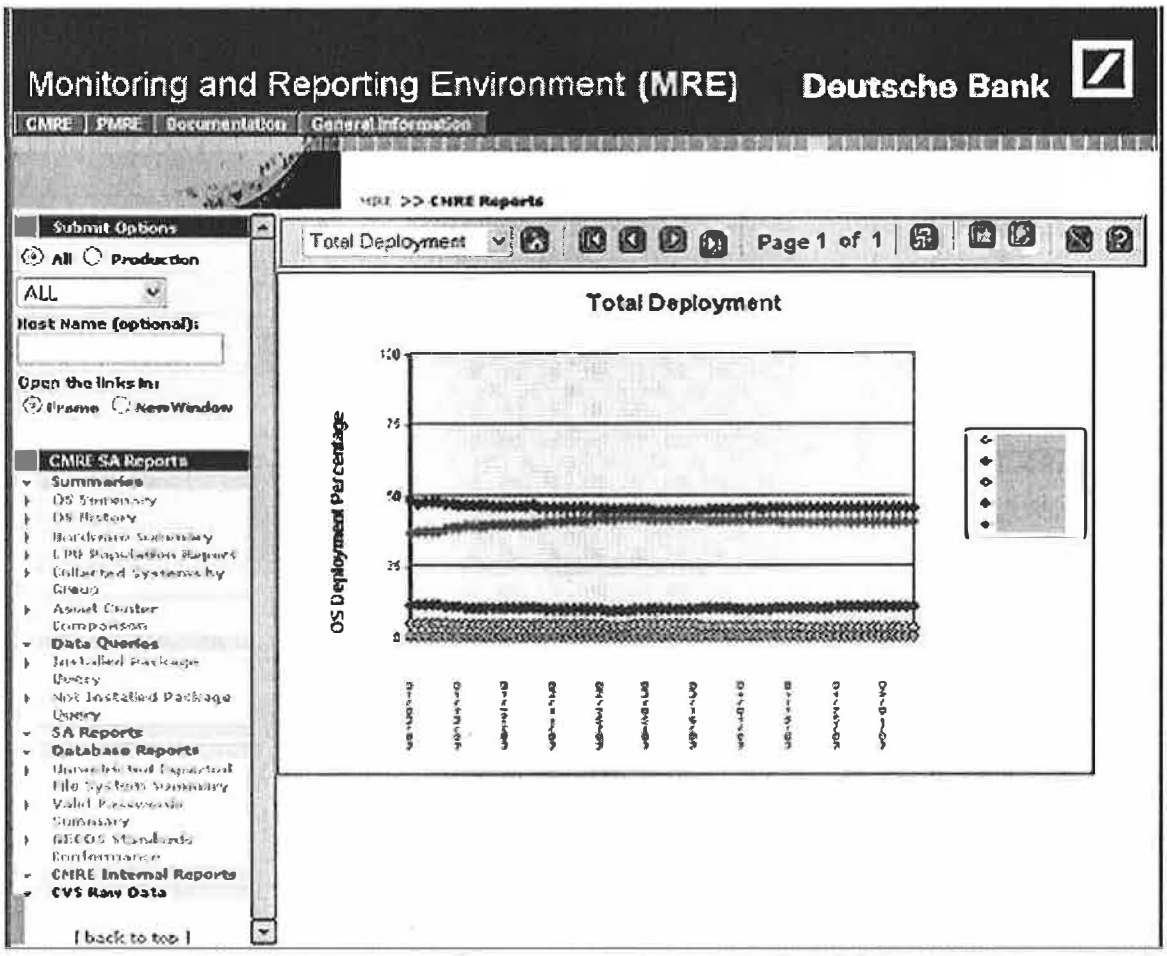


Figure 4: Sample of OS History Chart.

- Disk Information – Number, types, capacity, used capacity
- Passwd Information – contents of /etc/passwd
- Installed Packages and Patches
- Network Interfaces
- Services – both inetd.conf contents as well as netstat output

Processing is done via a series of custom Perl modules. Because the information we receive is typically from commands that deliver information in a human-readable form, we have to provide custom parsers for each type of output. The parsed data is uploaded into a database into tables that are designed for that particular data.

Reporting

Our initial implementation used custom reporting scripts. For each report we wished to present, we coded a CGI script in Perl to extract the data from the database or file system, and then present the report to the user. We wanted to create a more efficient and flexible process, therefore we examined a number of data mining products and selected one to replace a significant number of reports. This brought the development time of each report down to minutes from hours or days.

We provide a number of different types of reports. Some are simple aggregated data reports, such as the operating system summarization shown in Figure 2.

Note that all data in charts and graphs included in this paper are shown with either generated data created for demonstration purposes only or are redacted and presented to demonstrate format only.

We also provide charts of the same information (shown in Figure 2).

In addition we also collect historical information, so that we can see how we are doing over time, for instance how our migration to new operating systems is progressing (Figure 4).

Additional reports include queries such as all machines of a particular operating system that does or does not have a patch or package installed. This has come in useful in a number of occasions, for instance to track our rollout of Kerberos, to detect hosts that were running a susceptible version of sendmail, and to provide a list of systems that required SSH updates.

PMRE

In addition to collecting configuration information, we are also collecting performance information. We are currently using the public domain Orca program (http://www.orcaware.com/orca/) to collect data on all our systems. In addition to the standard (somewhat

Net Load Compared to Available Capacity

Hostname	Available Capacity	Percentage Utilized	Hostname	Net Load
foo1.uk.db.com	0.806	14.29%	foo2.us.db.com	0.660
foo2.us.db.com	0.740	47.14%	foo7.srv.uk.deuba.com	0.592
foo3.na.deuba.com	0.688	38.57%	foo4-adm.na.deuba.com	0.512
foo4-adm.na.deuba.com	0.608	45.71%	foo3.na.deuba.com	0.432
foo5.aus.deuba.com	0.564	40.00%	foo10.us.db.com	0.408
foo6.aus.deuba.com	0.538	42.86%	foo5.aus.deuba.com	0.403
foo7.srv.uk.deuba.com	0.528	52.86%	foo6.aus.deuba.com	0.376
foo8.srv.uk.deuba.com	0.464	34.29%	foo9.srv.uk.deuba.com	0.262
foo9.srv.uk.deuba.com	0.444	37.14%	foo8.srv.uk.deuba.com	0.242
foo10.us.db.com	0.432	48.57%	foo16.aus.deuba.com	0.215
foo11.srv.uk.deuba.com	0.403	14.29%	foo17.aus.deuba.com	0.202
foo12.uk.db.com	0.376	20.00%	foo14.uk.db.com	0.181
foo13.srv.uk.deuba.com	0.302	27.14%	foo23.uk.db.com	0.148
foo14.uk.db.com	0.289	38.57%	foo1.uk.db.com	0.134
foo15.uk.db.com	0.270	14.29%	foo13.srv.uk.deuba.com	0.112
foo16.aus.deuba.com	0.269	42.86%	foo22.apcc.ap	0.112
foo17.aus.deuba.com	0.255	45.71%	foo12.uk.db.com	0.094
foo18.uk.db.com	0.232	17.14%	foo11.srv.uk.deuba.com	0.067
foo19.sg.db.com	0.228	18.57%	foo19.sg.db.com	0.052
foo20.srv.uk.deuba.com	0.178	14.29%	foo18.uk.db.com	0.048
foo21.uk.db.com	0.175	15.71%	foo15.uk.db.com	0.045
foo22.apcc.ap	0.168	40.00%	foo21.uk.db.com	0.033
foo23.uk.db.com	0.132	52.86%	foo20.srv.uk.deuba.com	0.030
foo24.srv.uk.deuba.com	0.091	22.86%	foo24.srv.uk.deuba.com	0.027

Represents Demonstration Data Only

Table 4: Load can be shifted from busy systems to idle ones.

modified) graphs provided by Orca to allow people to see the performance of our system, we also do additional parsing of the data to determine overall weighted utilization numbers on our machines. This allows us to determine what our most and least used hosts are, globally. We can also drill down on individual machines. Currently we base our utilization number solely on CPU utilization, which usually proves to be an accurate representation. Clearly, in some cases we will drill down on a host and discover that while it is barely using any CPU, it is memory or I/O constrained, but this is the exception.

Once again, in addition to summarizations we provide host detail as well (Figure 5).

Problems Solved

CMRE has saved Deutsche Bank money in system retirements and cost avoidance, and it has reduced the number of people performing repetitive, manual tasks. Below are just some of the areas where having the information provided in CMRE has saved the bank money, time or both.

System Recovery

While lost system data can be recovered, a significant amount of specific machine configuration information is required in order to reconstruct the system fully after it has been lost (such as size and partitioning of disk, how much memory was installed, etc.). CMRE makes all this information easily available and accessible, so that rebuilding or replicating systems is a far simpler task.

More Efficient Reporting

Having security and audit vulnerabilities of our systems available in a single location, significantly reduces the time necessary for SA groups to track down and remediate problems.

System Retirements

Based on information presented in the utilization reports shown above, we were able to identify underutilized machines. As we were doing some major data center moves, we were able to retire a number of these

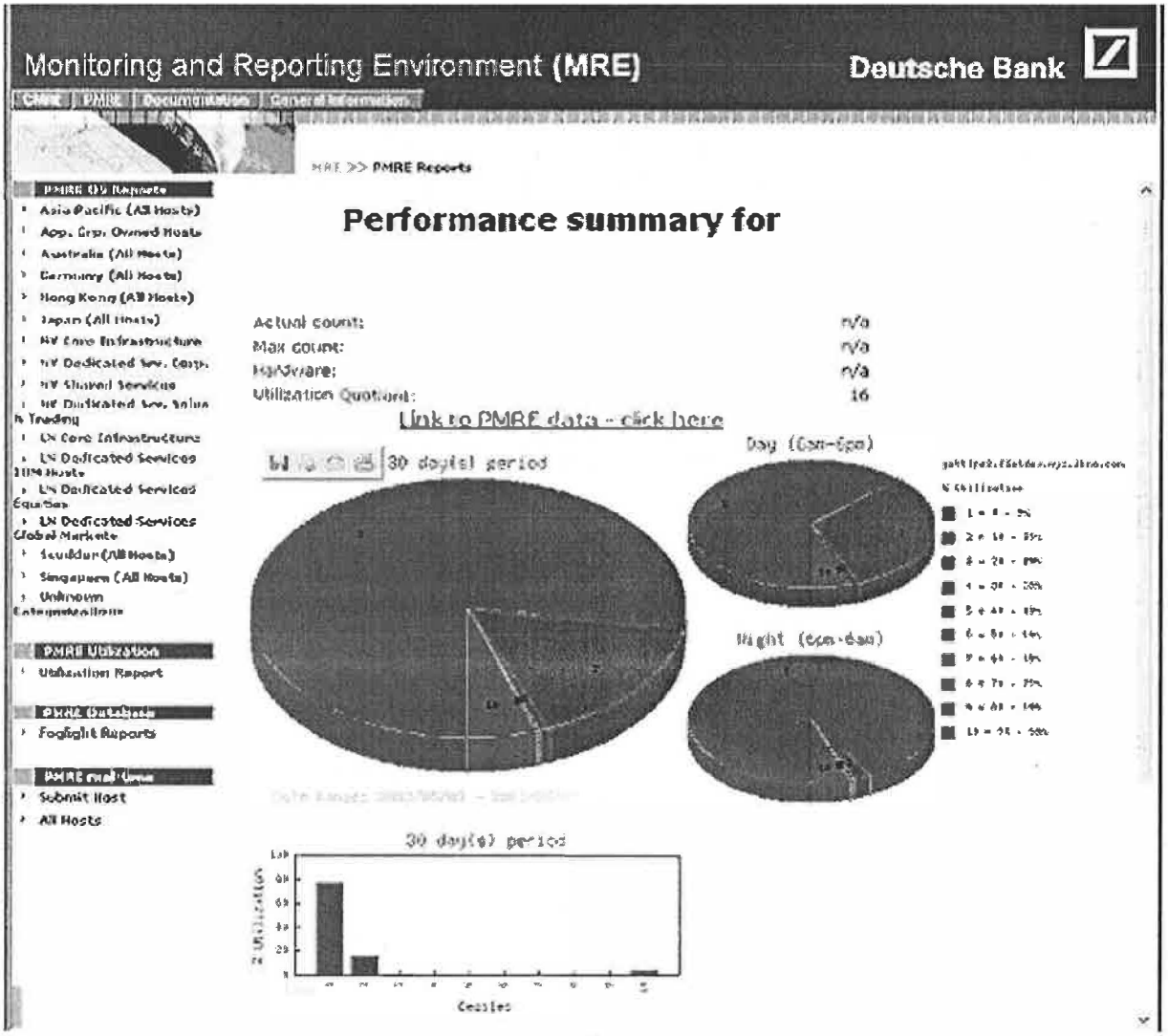


Figure 5: Host performance summary.

machines, thereby removing them from our inventory and avoiding costs of moving them.

Utilizing CMRE and PMRE, we were able to put together matrixes such as the one below to determine which systems could be retired. In the example below, the Available Capacity is a normalized number based on the speed and number of the processors (gathered with CMRE) minus the load currently on the system. The Net Load of a system is calculated based on a weighted CPU utilization formula over a 30 day period.

Table 4 shows instances where load could be shifted from a low utilized system (bottom to top of right column) onto another machine with excess capacity (top to bottom of left column) and then retire the resources. Additional reports provide information about depreciation of the machine (retrieved from our Asset database) and the cost of housing the machine in our data center.

Cost Avoidance

Even within large groups, systems are often purchased individually, with large amounts of space left for growth. The result is that systems may be significantly under populated. Figure 6 is report that a business area requested so that they could determine whether to purchase new machines (requiring additional data center space, network connections, etc.). The report showed that we had quite a few machines that were less than fully populated, and we could avoid the significant costs of purchasing new machines simply by purchasing additional CPU cards.

The Future of MRE

CMRE and PMRE are the first components of an overall management framework that we are calling MRE (Management and Reporting Environment). MRE is envisioned initially as a portal where all information about individual systems will be consolidated or viewable. Information available on MRE would include configuration, performance, asset information, security and event monitoring statistics, etc. For instance, someone investigating a performance issue could go to the single MRE portal first to look at the performance data returned, then to examine the configuration information that explainswhat may be causing the performance issues. Currently, MRE is a read-only system, because we are in the information gathering process. As we grow the various components of MRE and are able to tackle some of the challenges that lie ahead, we will be far better positioned for proactive systems management.

Availability

Because the system is made up of both commercial and non-commercial components, it cannot be made generally available as a package. However, the collector scripts as well as discussions and suggestions about the report formats are available by contacting the authors.

Author Information

Xev Gittler is the Regional Unix Architect for the Americas for Deutsche Bank. He has worked for nearly 20 years as a system administrator and system architect at a number of financial institutions, universities, and R & D labs. He is a past Vice President of SAGE as well as a founder and original president of NYSA, the NY Systems Administrators group. Xev would like to thank his wife, Rebecca Schore, for everything, but especially for helping turn this paper

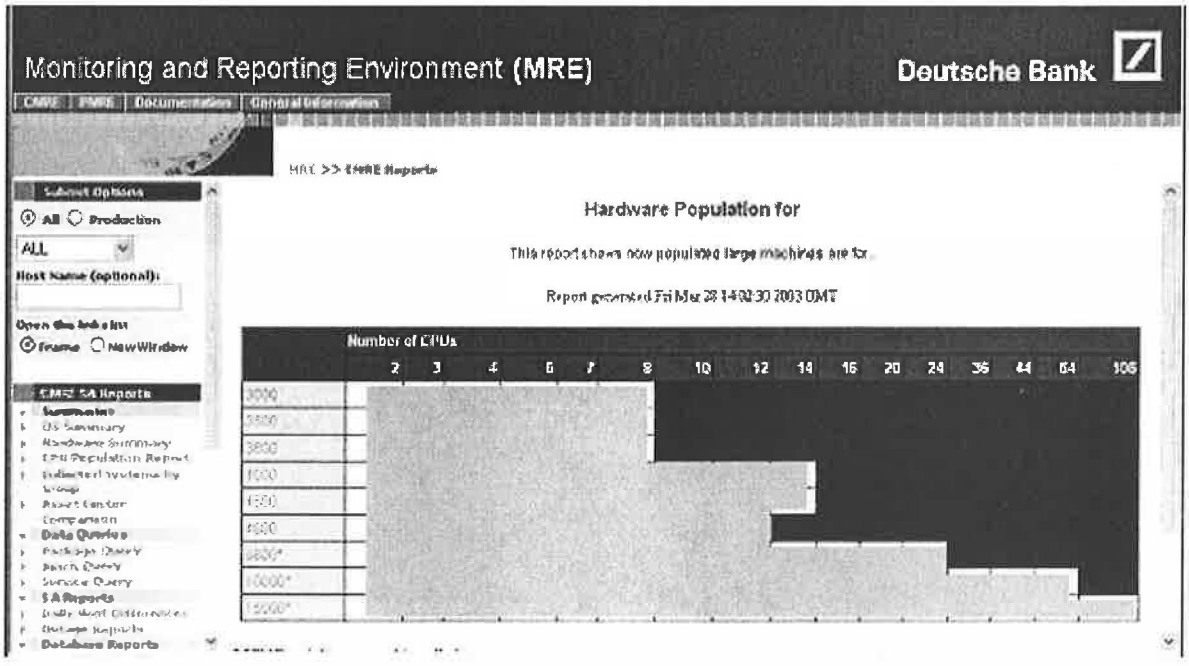


Figure 6: Sample hardware population report.

from ideas into something that could be put down on paper. Xev can be reached via email at Xev.Gittler@db.com.

Ken Beer graduated in 1992 from the University of Maryland with a degree in Philosophy. For the past 10 years he has been a Unix systems administrator in the financial sector in New York City. He is currently a VP of Unix Engineering at Deutsche Bank where he works on improving system maintainability. Ken lives in Brooklyn, NY where he spends any free time he can with his wife Cheri and son Satchel. Ken would like to thank his dad for buying him a subscription to Byte magazine in 1979. Ken can be reached via email at Ken.Beer@db.com.

References

Brooks, Frederick Phillips, *The Mythical Man-Month: Essays on Software Engineering*, Pearson Addison Wesley, 1988.

Gall, John, *Systemantics: The Underground Text of Systems Lore*, Sytemantics Press, 1986.

Perrow, Charles, *Normal Accidents: Living with High-Risk Technologies*, Princeton University Press, 1999.

Dorner, Dietrich, *The Logic of Failure*. Perseus Publishing, 1996.

Merton, Thomas, *The Unanticipated Consequences of Purposive Social Action*, 1936.

Hunt, Andrew, David Thomas, and Ward Cunningman *The Pragmatic Programmer From Journeyman to Master*, Addison-Wesley, 1999.

Burgess, Mark, "Computer Immunology," *LISA Conference Proceedings*, 1998.

Burgess, Mark, "A Site Configuration Engine," *USENIX Computing systems*, Vol. 8, Num. 3, 1995.

Traugott, Steve, "Bootstrapping an Infrastructure," *LISA Conference Proceedings*, 1998.

"A Simple Network Management Protocol (SNMP)," *RFC 1157*.

Oetiker, Tobias, "Template Tree II: The Post-Installation Setup Tool," *LISA Conference Proceedings*, 2001.

New NFS Tracing Tools and Techniques for System Analysis

Daniel Ellard and Margo Seltzer – Harvard University

ABSTRACT

Passive NFS traces provide an easy and unobtrusive way to measure, analyze, and gain an understanding of an NFS workload. Historically, such traces have been used primarily by file system researchers in an attempt to understand, categorize, and generalize file system workloads. However, because such traces provide a wealth of detailed information about how a specific system is actually used, they should also be of interest to system administrators. We introduce a new open-source toolkit for passively gathering and summarizing NFS traces and show how to use this toolkit to perform analyses that are difficult or impossible with existing tools.

Introduction

The purpose of most passive NFS tracing research tools is to discover and investigate workload characteristics that can be exploited by the operating system. The goals of the tools and methods described in this paper are quite different – we wish to use passive NFS trace analysis to help system administrators understand the workload on their systems, identify latent performance issues, and diagnose specific problems. For example, file system researchers might be content to know that on a particular system most of the reads come from a small number of large files, but the administrators of that system might also want to know the names of those files. If the system becomes swamped with requests for those files, the administrators might also want to identify the owner of the files and the users responsible for the requests so that those users can be consulted.

There already exist a number of tools that make it possible to quantify many aspects of NFS traffic, and system administrators have devised ways to use these tools in order to identify and debug common problems [17]. Unfortunately, most of these techniques are limited in their ability to analyze the activities of specific clients and users. Furthermore, they are tailored to solving specific problems and are difficult to use as a way to explore the wealth of information that can be gleaned from NFS traces.

In this paper, we describe nfsdump and nfsscan, a pair of tools that provide a framework for gathering and analyzing NFS traces in a general and extensible manner. nfsdump collects NFS traces, while nfsscan scans through the traces gathered by nfsdump and creates a set of summary tables that can be analyzed by simple scripts or imported into a spreadsheet or other data analysis tool for further processing. In addition to nfsdump and nfsscan, we describe ns_timeagg, ns_split, and ns_quickview, three applications that simplify manipulation of the tables created by nfsscan.

The advantage of decoupling the process of gathering traces from trace analysis is that the same traces can be analyzed several ways and at different time scales. It also means that there is a clear interface between these tools and that they can be used independently. For example, we have written several other programs to analyze the output of nfsdump and used them in our own studies of NFS workloads and usage patterns [6, 7].

Another advantage of the separation of the gathering and analysis of traces is that the data may be anonymized before analysis. The nfsdump toolkit includes an anonymizing postprocessor that anonymizes the client and server IP addresses, user and group ID numbers, and file names. All of the data presented in this paper have been anonymized so that they may be made public. In ordinary use by system administrators, however, this anonymization step would be omitted so that specific and potentially useful identifying information about users, groups, client hosts, and file names is preserved.

The rest of this paper is organized as follows:

- A discussion of related work and other tools for NFS analysis
- An overview of nfsdump and nfsscan
- A description of the systems used in our example analyses and case studies
- Basic analyses using nfsscan
- Case studies
- A discussion of the limits of passive tracing and how to augment nfsdump/nfsscan with active methods
- Conclusion

Related Work

NFS[1] trace analysis has a long and rich history as a tool for file system research. Because of this, most of the work in passive NFS trace analysis has focused on research topics such as characterizing workloads and investigating the effects of client-side caching [5, 15].

[1] In the context of this paper, NFS refers only to NFS version 2 protocol [14] and the NFS version 3 protocol [2]. nfsdump and nfsscan do not support version 4 of the NFS protocol [13, 16] or any variants of NFSv2 and NFSv3, although our tools may be extended to handle these protocols.

Matthew Blaze introduced several techniques for inferring the client workload from an NFS trace and implemented these techniques in rpcspy and nfstrace [1]. Andrew Moore gives a survey of these and other techniques and contrasts the analysis of file system workloads via passive network monitoring and direct kernel-level traces [12]. This work shows that passive NFS tracing reveals useful and accurate information about many aspects of an NFS workload.

There already exist many tools for gathering NFS traces or summary information about NFS activity. Basic tools like nfsstat(1M), iostat(1M), and nfswatch [4] provide aggregate statistics about NFS traffic. For many diagnostic purposes, these statistics suffice. Specialized tools such as nfstrace and the NFS Logging System [18] focus entirely on NFS. General-purpose network protocol monitors, such as tcpdump [11], ethereal [3], rpcspy [1], and snoop [19] can be used to analyze NFS traffic and other system activity. Most of these tools contain code to decode and print detailed information about NFS calls and responses, and therefore general-purpose tools like tcpdump have replaced special-purpose NFS monitors in many contexts.

In terms of general capability, our system most closely resembles nfswatch and the NFS logging/nfslogd system provided with Solaris. In comparison to nfswatch, nfsdump/nfsscan is more flexible because it runs on more platforms and captures more information – nfsscan can gather any combination of per-user, per-group, per-client, per-file, and per-directory information, while nfswatch gathers either per-user or per-client information.

In contrast to the NFS logging system and nfslogd, our system is completely passive, can run on a separate host instead of running on the server, and is portable across a range of platforms instead of being tied to Solaris. A shortcoming of nfsdump/nfsscan is that it does not attempt to reconstruct an application-level trace of the NFS activity in the same manner as nfslogd, but this capability could be added as a postprocess to the output of nfsdump (much as nfslogd postprocesses the records created by the NFS Logging System).

The Toolkit

Our toolkit consists primarily of a pair of tools, nfsdump and nfsscan. nfsdump collects NFS traces, while nfsscan takes those traces and generates summary tables useful for further analyses. In addition, we bundle three analysis applications in our toolkit – ns_timeagg, ns_split, and ns_quickview – which postprocess the nfsscan summary tables.

nfsdump

Our tracing system, nfsdump, is similar to nfstrace or tcpdump in general philosophy, but captures more information than either, and, unlike tcpdump, only decodes the NFS protocol. Like tcpdump, the output of nfsdump is human-readable text. nfsdump uses libpcap [10], the same packet-capture library as tcpdump, and has the ability to read and write raw packet files in the same format as tcpdump or any other tool that uses this format.

The most important functional differences between nfsdump and earlier tools are that nfsdump captures the effective UID and GID of each call, and properly decodes RPC over TCP (even with jumbo frames).

nfsscan

The first step of our analysis system, nfsscan, writes its data in a format designed to be easy to parse and interface with other tools. This is in contrast with most other tracing tools, which often generate output in an irregular, difficult to parse, or undocumented format – for example, tcpdump 3.7.2 prints file offsets for read and write operations in hexadecimal for NFSv3 requests and decimal for NFSv2 requests, but does not print whether the request is NFSv2 or NFSv3.

The output of nfsscan consists of one or more distinct tables (depending on how nfsscan is invoked). These tables can be analyzed by simple scripts or imported into a spreadsheet, database, or other data analysis tool for further processing. These tables include the following information:

- The total number of NFS operations and the number of times each NFS operation is called during each analysis period. The default analysis period is five minutes, but a different analysis period may be specified on the command line.
- The average latency of each type of NFS operation.
- A map of the file system hierarchy, and information about each file accessed.

  The file system hierarchy is inferred from the results of lookup, create, mkdir, rename, remove, and link calls. Information about each file is gathered from the responses to getattr and other calls.

By default, nfsscan only records the counts for the NFS operations that appear most frequently in a typical workload – read, write, lookup, getattr, access, create, and remove. Additional operations or an alternative list of operations may be specified on the command line.

Unless more information is requested, nfsscan creates per-server aggregate statistics in a manner similar to nfsstat. However, it can also subdivide the statistics by any combination of client, UID, GID, and file handle, and it can filter the data based on caller host, UID, or GID.

Helper Applications

The most general and powerful method for analyzing the tables created by nfsscan is to import these tables into a database or data analysis package. To simplify some of the more common analyses, however, our toolkit includes several helper applications that make it possible to manipulate the output of

nfsscan and perform useful analyses from the commandline or by using short shell scripts.

By default, nfsscan creates a single table containing aggregate operation counts for each five-minute interval of a trace. The user may specify a shorter or longer time interval, but this is usually unnecessary. A helper application named ns_timeagg makes it easy to compute aggregates across rows of the table created by nfsscan. For example, if the nfsscan time interval is five minutes, ns_timeagg can create a new table with a time interval of one hour directly from the output of nfsscan. It is typically several orders of magnitude faster to compute aggregates from the tables generated by nfsscan than it is to re-run nfsscan with different parameters. ns_timeagg can also aggregate by client host, UID, or GID.

In contrast to ns_timeagg, which combines rows from the table, ns_split provides a way to filter rows from the table, throwing away rows that do not match given criteria. A combination of ns_timeagg and ns_split is usually sufficient to isolate the interesting data. It is also easy to write scripts to manipulate the output of nfsscan, ns_timeagg, and ns_split because the tables are generated in a simple text format.

The helper application ns_quickview uses gnuplot [20] to create plots for the operation count tables created by nfsscan, ns_timeagg, or ns_split. ns_quickview can either save the gnuplot script to file, so that it can be edited before execution, or generate the plots immediately. ns_quickview also accepts commands to pass directly to gnuplot, which allows the user to customize the plots without editing the scripts. All of the plots in this paper were created by ns_quickview, using command-line options to tell gnuplot to create encapsulated postscript of the proper size and with the appropriate font.

We anticipate that for most purposes, nfsscan with the default parameters will provide enough information to identify general trends in the data. If the user discovers that a specific period of the trace is particularly interesting, he or she can re-run nfsscan with different parameters to extract additional per-client, per-user, per-group, per-file, or per-directory information, as described in the Case Studies section, to build a table to investigate specific questions. It is possible to simply create the full set of tables containing all of the information for the entire trace, but this requires a large amount of processing time and memory, and the resulting tables require a considerable amount of disk space.

Overview of the Example Systems

In this section we describe the computing environments from which we collected the traces that are used in later sections to illustrate the use of nfsdump, nfsscan, and the helper applications.

EECS03

The EECS03 traces were taken from the Network Appliances Filer that serves the home directories for most of the Computer Science Department at Harvard University from February 17, 2003 through February 21, 2003. EECS03 has two network ports, serving clients on different subnets. Our traces are gathered from one of these ports. The EECS03 traces do not include backup activity.

There is no standard EECS03 client machine, but a typical client is a workstation with at least 128 MB of RAM running GNU/Linux, UNIX, or Windows NT. The Windows machines access EECS03 via SAMBA running on a UNIX server, and therefore all of their network disk accesses appear in some form in the NFS trace. All workstations store their operating system and most of their applications on their local disk, and use their local disk for scratch space. EECS03 is used primarily for home directories and shared data.

Email for EECS03 users is delivered on a separate file system. Some users use procmail or other preprocessors that file their mail for them in their home directory, and this activity is reflected in our traces, but on the whole email appears to play only a small role in the EECS03 workload.

The EECS03 traces were taken from the same server as the EECS traces described in earlier work [6], but were taken more than a year later. Due in part to a redistribution of responsibilities among the servers of the EECS community, the workload of this server has changed significantly during this time. The largest difference is that the WWW server now uses EECS03 to store the main web pages; in EECS there was very little WWW traffic because most of the pages were stored on a separate file system. This raises the question of what the EECS03 workload would look like if we moved the WWW pages back to a separate file system. We will see how to investigate questions of this kind in the section "Exploring the Impact of the WWW Server on EECS03."

DEAS03

The DEAS03 traces were taken from the Network Appliances Filer that serves the home directories for the Division of Engineering and Applied Sciences at Harvard University from February 17, 2003 through February 21, 2003. The DEAS03 traces do not include backup activity. There is no standard DEAS03 client machine, but the typical DEAS03 client is a PC running GNU/Linux or Sparcstation running Solaris.

In contrast to EECS03, the DEAS03 workload does contain email. Email for DEAS03 users is delivered to a mail spool outside of the users home directory, but on the same NFS file system. In addition to email, the DEAS03 workload contains a fairly typical research and development workload.

The DEAS03 traces were taken from the same server as the DEAS traces described in earlier work [7], but in this paper we use a more recent trace.

ICE and MAIL

ICE and MAIL are traces from two different interfaces of the same Digital UNIX NFS server. This

machine, along with several similarly configured machines, hosts the home directories of the general-purpose accounts provided to all undergraduates, graduate students, and University staff. Each of these machines hosts several file systems and has several network interfaces, and each network interface communicates with a different set of clients. Backup activity is not included in either the ICE or MAIL traces, because a separate interface is used for backups.

The ICE trace captures the traffic from April 21, 2003 to April 25, 2003, between the NFS server and a set of machines that form the instructional computing environment, a shared resource used for computer science instruction and other academically-oriented work. Students are permitted to run email clients on the ICE machines. In this paper, we only analyze a trace of the traffic between one ICE host and one of the NFS home directory file systems.

The MAIL trace captures the traffic between the NFS server and the campus mail servers from May 5, 2003 to May 9, 2003. These machines host the IMAP and SMTP servers that handle the bulk of the campus email. Many users also login to these machines to run email programs, primarily pine. In this paper, we only analyze a trace of the traffic between one of the mail server hosts and one of the NFS home directory file systems.

The MAIL traces are taken from the same general environment as the CAMPUS traces from 2001 described in earlier work [6]. However, during the elapsed time the system architecture has changed significantly – the mail software has been changed, and nearly all of the client hosts have been replaced with new hardware running a different operating system. As a result, the MAIL workload is quite different from the CAMPUS workload.

Example Analyses

In this section, we provide examples of some of the analyses that are easy to perform with nfsdump/nfsscan, using five-day traces gathered from EECS03, DEAS03, ICE, and MAIL. Each of the traces begins at midnight on a Monday and continues until the end of the subsequent Friday. The traces have been processed with nfsscan, including per-client and per-user information in addition to the default information. We will assume that the output from nfsscan for each trace has been saved in files EECS03.ns, DEAS03.ns, ICE.ns, and MAIL.ns. For some of the examples, we will also assume that each of the .ns files has also been split into five files corresponding to the days of the trace, and given names like EECS03-1.ns.

What are the General Workload Characteristics?

The most general question of interest is how many operations of each type a server performs over a given period of time. We can use ns_timeagg to compute summary statistics for an entire table created by nfsscan by specifying a time interval of zero. Table 1 shows the total operation counts for five consecutive weekdays, and the percentage of the total contributed by each of the most common operations. Figure 1 shows the commands that generated the counts shown in Table 1.

```
ns_timeagg -t 0 EECS03.ns
ns_timeagg -t 0 DEAS03.ns
ns_timeagg -t 0 ICE.ns
ns_timeagg -t 0 MAIL.ns
```

Figure 1: Commands to generate the data for Table 1.

Table 1 illustrates the diversity of workloads that NFS servers must support:

Read/Write Ratio All of the systems have a read/write ratio of more than one. EECS03 has a read/write ratio of approximately 2, while DEAS03 has a ratio of 3, and ICE has ratio of more than 4. MAIL, on the other hand, is utterly dominated by reads. More than 93% of the operations in the MAIL trace are reads.

Data and Metadata The EECS03 and especially ICE workloads have more metadata requests than reads and writes – there are more requests for information *about* files or directories than requests to read and write the contents of files. However, the operation mixes differ – EECS03 is dominated by access and lookup calls, while ICE also has a large number of getattr calls.

In contrast, DEAS03 and MAIL are dominated by data operations, particularly read.

When is the System Most and Least Busy?

In order to plan maintenance activity or schedule large jobs in such a way to make the best use of an NFS server, it is important to know when the system is busy and whether there are idle periods. In our own research we have found that on many systems the load varies in a highly predictable manner according to the time of day and day of week. However, these patterns are not always intuitive – on one system, we found that the system was swamped with requests between 4:00 am and 9:00 am because several users thought that this would be the time when the system would be

System	Total Ops	read	write	lookup	getattr	access	create	remove
EECS03	33639133	22.27%	10.94%	28.90%	1.73%	22.83%	0.72%	0.86%
DEAS03	134603789	50.03%	17.70%	3.35%	25.63%	1.49%	0.26%	0.39%
ICE	37543922	3.11%	0.74%	16.30%	33.21%	39.21%	0.01%	0.00%
MAIL	626452462	93.39%	0.36%	1.82%	2.44%	0.65%	0.15%	0.29%

Table 1: Total operation count for five consecutive weekdays, and percent of the total for the most common operations.

most idle, and therefore they each scheduled their largest jobs to run at this time.

We can use ns_timeagg to aggregate the data by client, and then use ns_quickview to create plots of the data. Note that for a quick look at the basic load patterns, a five-minute interval is too short. To change the interval, use the -t flag, which takes a parameter measured in seconds. An example of a command to create a quick plot of the total operation count of EECS03 for each 30-minute interval during the trace period is shown in Figure 2.

```
ns_timeagg -t 1800 EECS03.ns \
           > EECS03.tmp
ns_quickview EECS03.tmp
```

Figure 2: Commands to create and display a plot of total operation counts of EECS03 over the five days summarized in EECS03.ns.

```
ns_timeagg -t 1800 EECS03-1.ns \
           > EECS03-1.tmp
...
ns_timeagg -t 1800 EECS03-5.ns \
           > EECS03-5.tmp
ns_quickview -l EECS03-[12345].tmp
```

Figure 3: Commands to create and display the overlay plot of total operation of EECS over the five days summarized in EECS03.ns.

Figure 4 shows examples of plots created by ns_quickview. The left column of this figure shows plots of the total operation count for each host for each half hour period over five consecutive weekdays. The right column shows the same data, but with the data for each day plotted on top of each other.

The plot from the commands in Figure 2 is shown in the top left of Figure 4. To see if the workload repeats strongly from one day to the next, we can take each day of the five day period and use ns_quickview to overlay the data on the same hourly scale. The command for accomplishing this is shown in Figure 3, and the resulting plot is shown in the top right of Figure 4. We repeat this pair of plots for the remaining systems (MAIL, ICE, and DEAS03). Daily cycles, if any, will be readily apparent.

MAIL, ICE and DEAS03 have clear daily rhythms. For all three systems, the operation rate is low in the late evening and early morning, and then climbs rapidly though the morning, reaching a peak at approximately noon, and then staying high throughout the afternoon. The operation rate for DEAS03 decreases rapidly at the end of business hours, but ICE and MAIL are still busy until midnight. Both systems have predictable periods of relatively light activity.

In the plot for the entire week for EECS03, it is hard to see any particular patterns, but the overlay plot shows that in addition to a general increase of activity during business hours, there are at least three regular daily events that cause load spikes twice in the early morning, and once in the late evening. The regularity of these events suggests that they are probably cron jobs.

The overlay plot for ICE shows that the workload is remarkably consistent from one day to the next, particularly in the early morning hours. We also note that the workload is lower than average for the weekdays on Friday afternoon and evening. (This is not apparent from the plots as they appear in this paper, because the key that shows which line corresponds to each day has been omitted from these plots. If requested, ns_quickview will label each line.)

Note that the first day of the traces for EECS03 and DEAS03 is a University holiday, and this appears to have an impact on both systems. Also note that there is a gap in the MAIL traces during the evening of Thursday 5/9/2003 due to a problem with the host that gathered the traces.

```
ns_timeagg -t 0 -B C EECS03.ns | \
      sort -k7 -n -r > EECS03.cli.tmp
```

Figure 5: A command to find the per-client operation counts for all of the clients of EECS03, sorted in descending order by total operation count. Note that column 7 of the output is the total operation count.

Which Clients Are Busiest?

Another question is which clients contribute the largest load to the system over time. We can investigate this by using ns_timeagg to compute the per-client operation counts. Figure 5 shows a command that computes the cumulative per-client operation counts for the entire trace period, and then sorts the resulting table in descending order by total operation count to find the busiest clients. The ten busiest clients for EECS03 are listed in Table 2.

Once we have identified the busiest clients (or any other clients that we think are interesting) we can use ns_split to extract the contribution from those clients, use ns_timeagg to create a table of the activity of those clients over time, and then use ns_quickview to create plots from these tables, using a command like the one shown in Figure 6.

```
ns_split -B C -c 0.0.0.51 EECS.ns | \
   ns_timeagg -B C -t 1800 > EECS03.tmp
ns_quickview EECS03.tmp
```

Figure 6: Commands to plot the operation count of client 0.0.0.51.

Figure 7 shows plots generated via ns_quickview for the four busiest systems.

Which Users Are Busiest?

Finding the busiest users is done in the same manner as finding the busiest clients. The only difference is that the aggregation is per user, instead of per client.

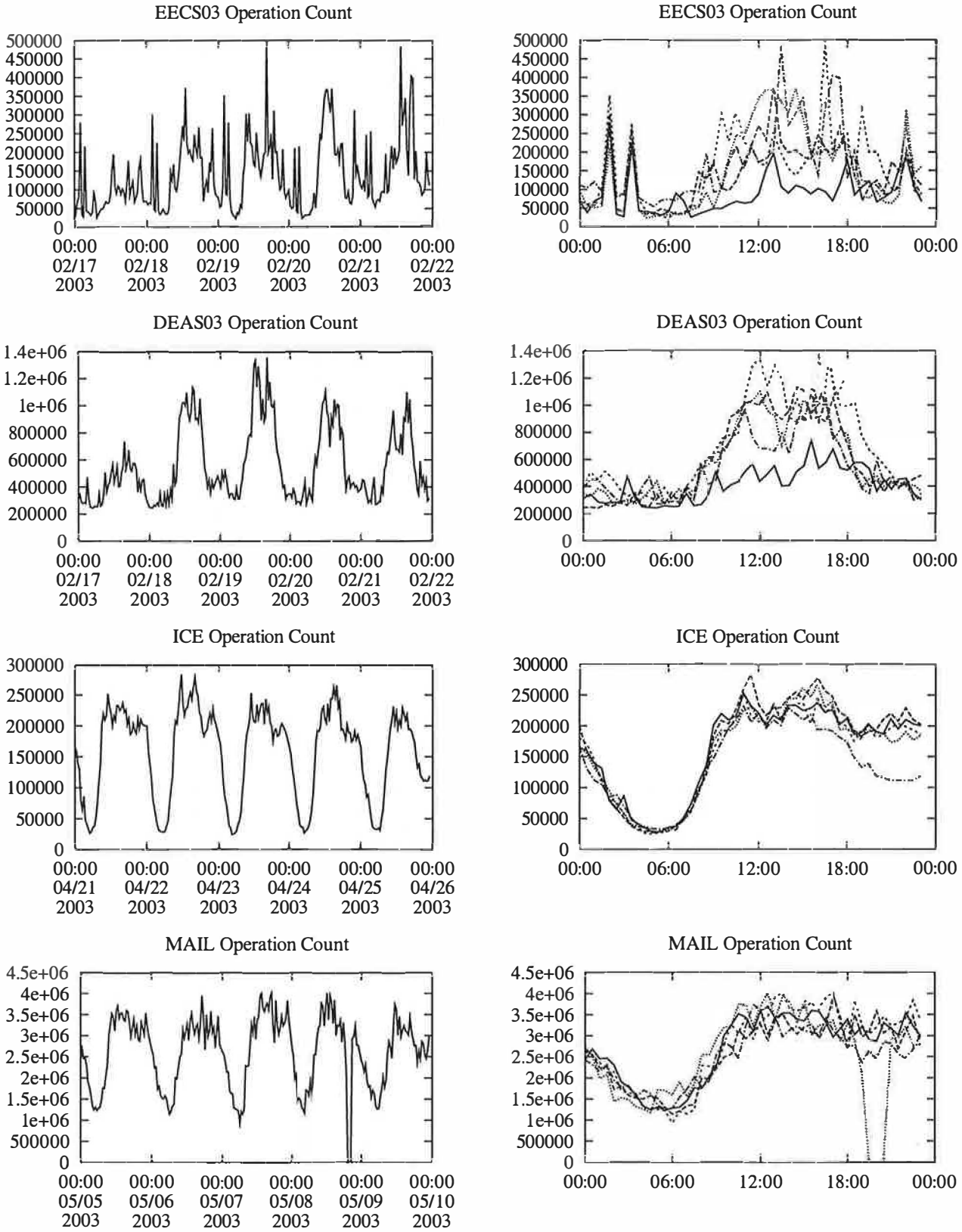


Figure 4: Plots of the total operation counts for each 30-minute period of five consecutive weekdays. Each plot in the left column shows the total operation count per half hour period for the entire five days. Each plot on the right shows the same plot, but with the data for each day superimposed on the others.

A table of the operation counts for the ten busiest users on EECS03 for the five days 2/17/2003 through 2/21/2003 is shown in Table 3. Once again, there is great diversity of workload patterns illustrated in this simple table.

- The busiest "user" is the www user. The operation counts for this user are dominated by information requests, but also contain some reading, and very little writing.
- The second busiest user is a user whose operation count is dominated almost entirely by reading.
- The third busiest user is the user of workstation 0.0.0.53, one of the busiest clients.
- The fourth busiest "user" is root, and the operation counts for root contain almost no reads and writes. The fact that root accounts for much of the workload is surprising. We investigate the causes of this behavior in the next section.
- The fifth busiest user is the user of workstation 0.0.0.130, another one of the busiest clients.

What Files are Busiest?

We can also use nfsscan to discover which files and directories are the object of the most operations, and what those operations are. This can be useful to identify hot files or directories, which might benefit from being replicated (if read-only), moved to a local disk (if not shared), or moved to a faster file server.

nfsscan can compute per-file operation counts in the same manner as it can create per-user and per-client operation counts. It can also gather additional descriptive information about files, such as their pathname, owner, permissions, and modification times. To save space, particularly when per-user or per-client information is gathered in addition to per-file information, the file description information is stored in a separate table. Both tables can be indexed by file handle, however, so they can be joined if necessary.

Which Directories are Busiest?

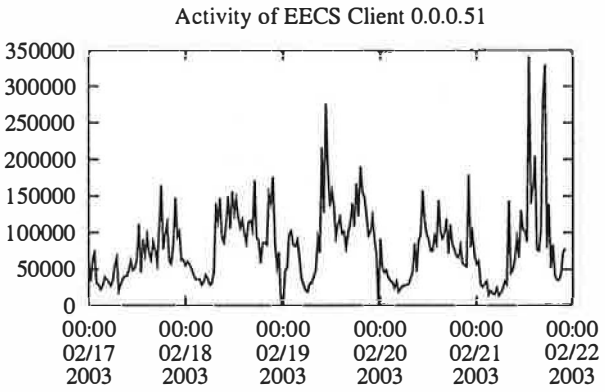
nfsscan can also compute per-directory operation counts. The directory operation count is defined as the sum of the operation counts for all of the files and subdirectories of the directory. This metric is useful for identifying collections of files or directory hierarchies that are the object of many operations even if each individual file is not the object of many. For example,

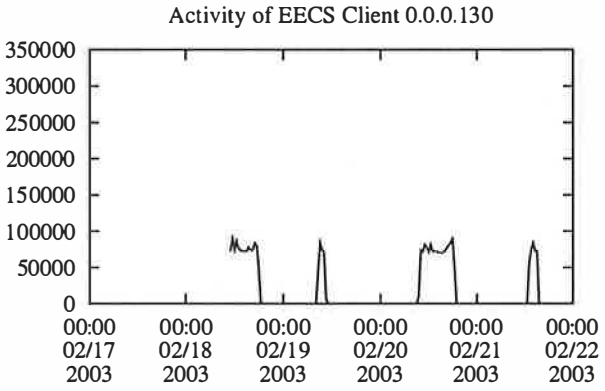
Client ID	Total Ops	Read	Write	Other	System Use
Total	33639133	7491050	3680255	22467828	
0.0.0.51	19560815	4009983	1489135	14061697	Mail and WWW server
0.0.0.130	3106796	38633	21444	3046719	Desktop of user1
0.0.0.53	2847486	460192	580911	1806383	WWW server/Desktop of user2
0.0.0.80	1304624	20778	644105	639741	Desktop of user3
0.0.0.84	938404	279071	362768	296565	Cycle server
0.0.0.56	668044	13379	95665	559000	SunRay server
0.0.0.68	614755	128977	99706	386072	Desktop/Cycle server
0.0.0.54	528469	501601	3786	23082	Cycle server
0.0.0.55	519653	492413	3728	23512	Cycle server
0.0.0.57	514069	487531	3705	22833	Cycle server

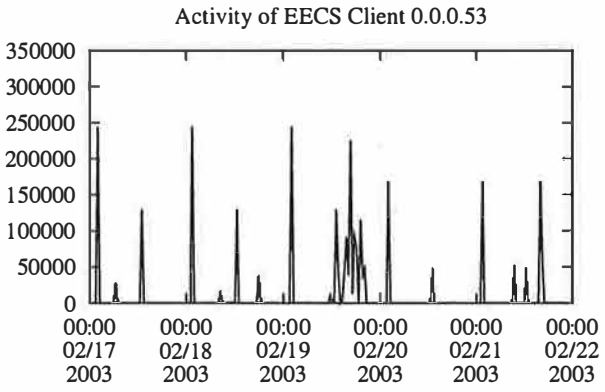
Table 2: Total operation counts for the ten busiest EECS03 clients for the period 2/17-2/21/2003. Note that the client identifiers have been anonymized.

User ID	Total Ops	Read	Write	Other	Role
Total	33639133	7491050	3680255	22467828	
101002	7683078	1268777	1024	6413277	WWW
101000	2494411	2396508	14917	82986	Grad Student
101009	2461584	384792	496133	1580659	Grad Student (user 2)
0	2020323	16	48	2020259	Superuser (root)
101060	1659601	70247	25455	1563899	Faculty Member (user 1)
101022	1371083	178598	13357	1179128	Faculty Member
101062	1310198	21422	644776	644000	Researcher (user 3)
101035	1131660	96183	1727	1033750	Grad Student
101001	1076060	293780	366905	415375	Grad Student
101034	1008195	138130	817268	5279	Grad Student

Table 3: Total operation counts for the ten busiest EECS03 users for the period 2/17-2/21/2003. Note that the user ID numbers have been anonymized, except for www and root. Users user1, user2, and user3 are the same users referenced in Table 2.







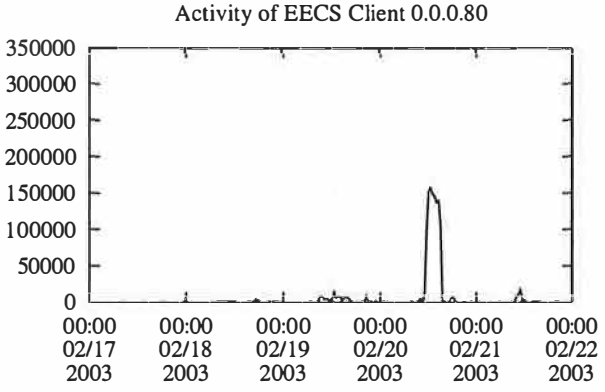
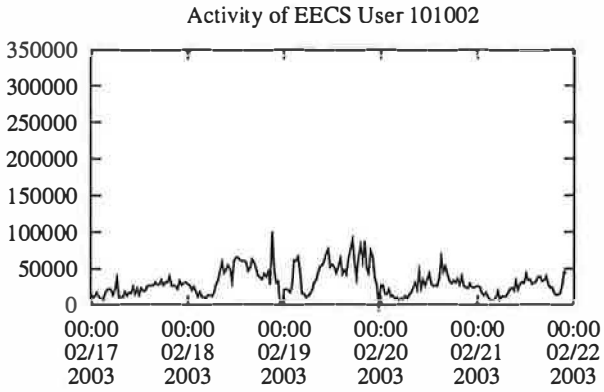
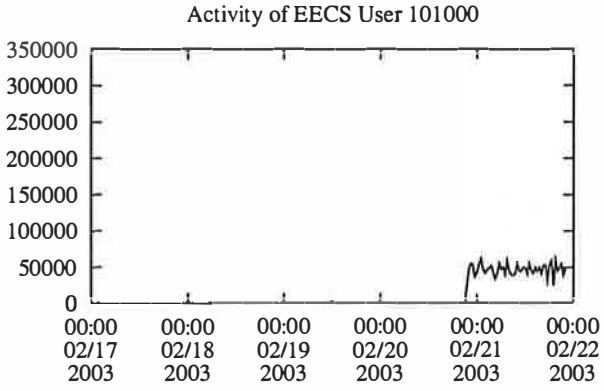
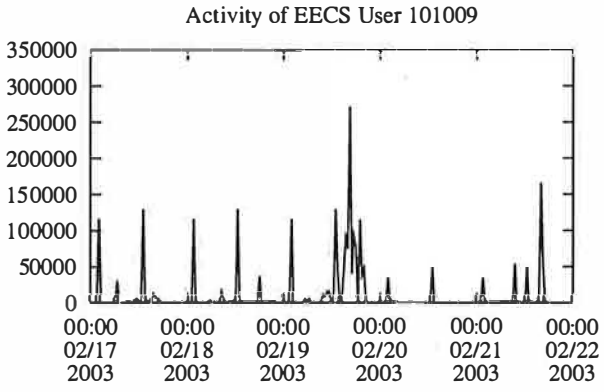


Figure 7: Plots of the total operation counts for each 30-minute period of five consecutive weekdays for the four busiest EECS03 clients.







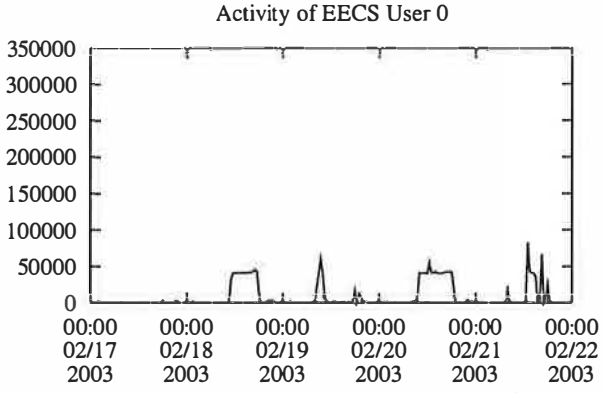


Figure 8: Plots of the total operation counts for each 30-minute period of five consecutive weekdays for the four busiest EECS03 users.

some mail clients store each email message as a separate file in a directory dedicated to this purpose, while others store an entire mailbox (or even a collection of mailboxes) as a single file. In the latter case, it is straightforward to identify and count all of the operations associated with the email of a particular user by keeping track of all operations associated with a single file. In the former case, however, it is much more difficult because the activity is distributed over a directory of files. In this situation, per-directory operation counts are a useful way to aggregate all of this activity into a single table row.

Another example where per-directory operation counts can be helpful is for measuring the total usage of shared directories, such as source code repositories.

Per-directory operation counts can also help identify the busiest subdirectories of a web site. Although it might seem that much of this information can be inferred from the web server logs (which record details about what requests web clients make, and how much data the web server sends to its clients), this may lead to inaccurate conclusions. It may be that there are important differences between the traffic from a web server and its clients and the traffic from the web server and the NFS server. For example, if most requests to the web site are for the same pages, these pages may be cached in the web server and therefore these requests will not translate into much NFS activity.

As another example, typical web server logs contain no useful information about what files are accessed each time a dynamically generated web page is visited. The total amount of data used to generate such a page might be very different from the amount of data that the web server finally sends to the client.

Case Studies

Previously, we showed how to use nfsscan, ns_timeagg, ns_split, and ns_quickview to perform several basic analyses. In this section, we show how to perform deeper analyses by exploring several interesting aspects of the traces.

What is Root Doing on EECS03?

As we saw in Table 3, root was one of the busiest users on EECS03 during the trace period. This is surprising because the EECS03 server is configured to treat root as an untrusted user except from a small number of machines. It is also interesting that nearly all of the NFS operations performed by root during this period are not operations that nfsscan ordinarily considers interesting, and therefore does not tabulate.

To investigate this mystery, we first used ns_split to remove the "trusted" hosts from the table. This made little difference, however, which was a bit troubling because this appears to imply that at least one supposedly untrusted host was able to access the server as root. This, of course, is the kind of situation that haunts the nightmares of NFS system administrators, and so we immediately investigated further.

The plots for the load from client 0.0.0.130 (in Figure 7) and the load for the root user (in Figure 8) provided a valuable clue. These two plots are very similar in shape (although not in magnitude), suggesting that the activity of this host and the activity of the root account are linked. A quick check using ns_split to isolate the contributions of the root user on client 0.0.0.130 confirmed that most of the operations performed by root during the trace period were indeed taking place on that system. Using ns_split again to isolate the activity of the user who owns client 0.0.0.130, we observe that at every time scale, plots of the total operation count for this user and for root have an almost identical shape. A further mystery was that the operations performed by the user mostly fell into the category of "normal" operations, unlike virtually all of the operations performed by root.

Now that we knew exactly what host to check and what period of the trace to examine, we re-ran nfsscan on a short section of the EECS03 traces, asking it to collect operation counts for all the NFS operations instead of only the default set of "interesting" operations. We discovered that all of the mystery operations were fsstat calls. Looking at the trace itself showed that for every operation invoked by the user, there was one or more corresponding fsstat call.

The fsstat call requests information about a NFS file system, including how much free space there is, and whether there have been any changes to the underlying file system (such as might be caused by a reboot or remount). Because fsstat does not access information about specific files or directories, it is permitted to run as root even if root is untrusted, and many NFS client implementations, including the one running on 0.0.0.130, are implemented to treat fsstat calls as if they were being performed on behalf of root.

The reason that 0.0.0.130 stands out is that the NFS client used by this host is implemented in a very conservative manner, and during the trace period was also misconfigured, and therefore used fsstat incessantly, generating at least one fsstat call per ordinary operation. The fact that the primary user of client 0.0.0.130 is also one of the busiest users in the trace makes this behavior stand out even more.

Unfortunately for the purpose of our analysis, the user of host 0.0.0.130 noticed that the machine seemed sluggish and requested an upgrade of the OS before we had an opportunity to discover the cause of the excessive fsstat calls. However, using our tools we were able to confirm that during later trace periods the number of fsstat calls generated by host 0.0.0.130 did return to normal levels.

The behavior exhibited by client 0.0.0.130 raises some administration issues. When confined to a single

host, the "extra" fsstat calls have no serious implications, but this behavior could become a problem if the EECS administrators had deployed this configuration of this NFS client implementation across the entire department. If *all* of the clients of a server exhibited this behavior, the number of NFS requests would nearly double. On a relatively unloaded server (such as EECS during the EECS03 trace period) this additional load might not cause a problem (or even be noticed), but on a more heavily loaded system, such as DEAS03, ICE, and especially MAIL, doubling the number of NFS requests could have an impact on the total system because the additional requests would increase the apparent latency and overall utilization of the network.

Exploring the Impact of the WWW Server on EECS03

We noted in the previous section that the busiest EECS03 user is the WWW server account, and that the busiest client hosts the department web server.

We also noted that the EECS03 trace exhibits different load characteristics from an earlier EECS trace, and theorized that this might be due to the fact that the earlier trace did not contain accesses to the WWW site. With nfsscan, we can test this theory.

Two of the defining characteristics of the EECS traces are a read/write ratio of less than 1, and that requests for metadata outnumber reads and writes significantly. Since the WWW server account is by far the busiest user, and is dominated by reads, a reasonable hypothesis is that the EECS03 workload would resemble the EECS workload if the WWW server traffic was removed. We can test this hypothesis by using ns_split to create a new table with all of the contributions from the WWW server account removed.

Removing the WWW server account from the workload does not change the character of the workload very much. The new workload still has a read/write ratio that is much closer to that of EECS03 than EECS. However, we can also note that after the WWW server account is removed, the busiest remaining user (user 10100) also has a notably high read/write ratio. Furthermore, if we use ns_split to make a table consisting only of this single user, we can see that the load from this user is distributed over a handful of machines. From our knowledge of the role of each of these machines, and our familiarity with the activities of this user, we know that the workload we are seeing from this user is probably due to this user running several large analyses.

While it is normal for users to run their analyses on these machines, it is arguable that this analysis might be unusual because it does not resemble things we have seen in other traces. We can use ns_split again on the original table to compute the operation counts with both the WWW server and this user removed. The resulting operation counts do resemble the counts from the EECS traces – the read/write ratio is approximately 1, and requests for metadata (lookup, getattr, and access) dominate requests to read and write data. Therefore, if we can successfully argue that the high operation counts for user 10100 during this period are unusual, then we can claim that the workload of EECS03 is similar to EECS with the addition of the workload contributed by WWW server.

However, there is an additional wrinkle to the analysis. In the original EECS configuration, the department web pages were hosted on a separate file system, but user home pages and some project home pages were hosted on the EECS NFS server. Therefore, accesses to personal home pages were recorded in the EECS traces, but they had little effect on the total overall workload. Using the per-directory statistics generated by nfsscan, we can see that this is no longer true (at least for this trace period). During this trace, project and user home pages contribute the overwhelming portion of the web-oriented traffic. Therefore, we can conclude that moving the departmental home pages back to a separate server would have little impact on the EECS03 workload. Moving user and project home pages to another server, however, would significantly reduce the EECS03 workload.

Why is MAIL So Busy?

The MAIL workload is prodigious, and is almost entirely reads. What is the source of all of these reads?

Using nfsscan with per-user and per-file information, we can quickly identify the busiest files in the system, and see that nearly all of the reads come from mailboxes.

Examination of a short trace (2:00 pm-3:00 pm on 5/6/2003) with nfsscan shows activity to 1033 inboxes on the MAIL file system. Approximately 70 of the owners of these inboxes have chosen to forward their email to other accounts or use an email preprocessor to categorize their incoming email, and therefore have very small inboxes. For the rest of the users, the median inbox size is more than 7 MB, and the largest 10% are over 35 MB. To make matters worse, these files are apparently read and re-read repeatedly by the email clients.

We believe that much of the problem is due to the particular properties of NFS client caching. NFS semantics permit the client to cache data read from a file, but provide only weak constraints on the consistency of cached data. The server does not know whether the client is caching data, and does not inform the client if the data is updated by another writer. Instead, the server relies upon each client to check periodically whether its cached data are up to date by asking the server whether the underlying file has changed. To make matters worse, the client can only ask whether the *file* has changed, and not individual blocks. This means that any change to a file (even rewriting a single byte with the same value) will either

cause NFS to invalidate all cached data associated with that file, or permit an inconsistency between the state of the file on the server and the client.

In order to prevent inconsistencies and scrambled mailboxes, the mail clients and email servers running on the MAIL client NFS hosts essentially disables client-side caching. Combined with frequent mailbox scans performed by the mail clients and the large size of the mailboxes, the system is doomed to a heavy workload of incessant reads. Therefore we believe that accessing flat-file mailboxes over NFS is simply not practical in an environment such as MAIL. Elprin & Parno [8] have shown that IMAP servers that store email in databases can be significantly more efficient than servers that store their email in flat files, and we believe that such systems will be widely deployed as mail workloads continue to grow.

Active Analyses

One of the most attractive properties of collecting traces via nfsdump (or most of the the other tracing tools) is that it is completely non-invasive and unobtrusive; it requires no changes to the client or server, nor does it place any additional load on the system. However, there are hard limits to what can be learned from passive tracing because we can only observe active files and directories – if a file or directory is never accessed, then no information about it will appear in the trace. We can augment passive techniques with a small amount of active probing to the workload in order to discover information that would otherwise remain hidden.

There are two mechanisms by which NFS requests specify the file or directory to act upon. Requests such as read and write specify the file via a *file handle*, an opaque object provided by the server to uniquely identify a file. Requests such as create or rename, on the other hand, specify a file via its name and the file handle of its parent directory. It is often useful to have a complete mapping between file names and file handles, so that we can identify all the operations that touch a particular file or directory, whichever way the identity of the file or directory is specified. We can infer most of this map simply by observing the results of the operations that the clients use to traverse the file system. Unfortunately, there are three cases in which we will fail to discover the proper mappings for a particular file or directory:

- The file or directory is never accessed.
  The easiest method for making sure that all files are observed is simply to run a process that traverses the entire file system. We can use the techniques discussed previously to find a time when we believe that the system will be relatively quiet and then use a command like find to traverse the file system completely.
- The file or directory is accessed, but only using the file handle or name. We never see the lookup request that connects the name to the file handle.
  This can happen when the clients cache the directory data for the parent directories. If the directories do not change, the client can do the lookup out of its own cache, and we will not have the opportunity to observe it. This appears to often be the case for directories near the root of the file system, or directories that are particularly busy. This is troublesome because these are the directories that are usually the most interesting.
  To make sure that these files and directories are observed, we need to find a client host that does not have these directories cached, and then perform a shallow scan of the file system by using find with a small maxdepth. The easiest way to accomplish this is to have a client unmount and remount the file system (but this is disruptive to any users on that client, so it is best to use a client host that is otherwise idle).
- The file is accessed, but the requests or the corresponding responses are omitted from the trace because of network errors or other problems.
  In this situation, the only practical solution is to be patient and hope that the file will be accessed again soon.

Ideally, it would be nice to run full traversals constantly in order to maximize the file system information captured by each trace. For most systems, however, this is simply not practical, because a full traversal can place a significant load on the system and on large file systems may require hours to run. In contrast, a shallow traversal requires a small fraction of the time of a full traversal, and captures information about the hottest directories.

Conclusion

We have presented nfsdump and nfsscan, a framework for gathering NFS traces and performing simple analyses and shown how to use these tools to perform various analyses of NFS activity, including investigating aspects of the workload of four production NFS servers.

Status and Availability

The source for nfsdump, nfsscan, and the related tools mentioned in this paper are available for download from http://www.eecs.harvard.edu/sos/software/.

Acknowledgments

This work would not have been possible without the contributions of many people, especially Peg Schafer, Aaron Mandel, Chris Palmer, Lars Kellog-Stedman, Scott McGrath, and Alan Sundell, who allowed me to gather traces from NFS servers under their administration. Peg Schafer, Lois Bennet, and Alan Sundell also provided suggestions for useful analyses. Our paper shepherd, Mario Obejas, provided helpful comments and suggestions regarding the structure and content of this paper.

This work was funded in part by IBM.

About the Authors

Daniel Ellard is a graduate student at Harvard University, where he expects to complete his Ph.D. in Computer Science this year. His thesis research concerns self-tuning file systems that automatically learn heuristics for choosing appropriate layout and replication policies for new files based on the observed access patterns of existing files. His research interests also include distributed systems and pedagogical techniques for introductory computer science courses. Before starting his Ph.D. research, Daniel was employed by BBN Laboratories for more than ten years, where he wrote software for planning and logistics, speech recognition and modeling, undersea warfare simulation, parallel programming tools, and Chrysalis, SunOS, and pSOS device drivers and programming interfaces for array processors, tape drives, and communication peripherals. Daniel Ellard can be contacted at ellard@eecs.harvard.edu.

Margo Seltzer is a Gordon McKay Professor of Computer Science and Associate Dean for Computer Science and Engineering in the Division of Engineering and Applied Sciences at Harvard University. Her research interests include file systems, databases, and transaction processing systems. She is the author of several widely-used software packages including database and transaction libraries and the 4.4BSD log-structured file system. Dr. Seltzer spent several years working at startup companies designing and implementing file systems and transaction processing software and designing microprocessors. She is a Sloan Foundation Fellow in Computer Science, a Bunting Fellow, and was the recipient of the 1996 Radcliffe Junior Faculty Fellowship and the University of California Microelectronics Scholarship. She is recognized as an outstanding teacher and won the Phi Beta Kappa teaching award in 1996 and the Abrahmson Teaching Award in 1999. Dr. Seltzer received an A. B. degree in Applied Mathematics from Harvard/Radcliffe College in 1983 and a Ph.D. in Computer Science from the University of California, Berkeley, in 1992.

Future Work

nfsdump and nfsscan are works in progress. We expect that they will evolve as users experiment with them and provide feedback or requests for new functionality. In particular, we hope that users will share whatever scripts they develop to encapsulate common analyses or administrative tasks so that we may add them to the nfsdump/nfsscan distribution.

We have also identified several specific areas for future work:

- nfsscan could be implemented in a more efficient manner. For heavy workloads such as MAIL, nfsscan requires all of a fast processor just to keep up with nfsdump. Since much of the processing time of nfsscan is spent parsing its input, it may be necessary to change the output format of nfsdump in order to increase the speed of nfsscan.
- nfsscan could extract the inode number from file handles for popular server types. The inode number is more useful than the file handle for tracking down specific files or directories.
- We would like to find a way for nfsscan's method of translating file handles into paths (by inferring the file system hierarchy from the results of lookup, create, and rename operations) to work gracefully with the NetApp snapshot mechanism [9]. It is not clear whether there is any way to resolve this problem in a general manner.

References

[1] Blaze, Matthew A., "NFS Tracing by Passive Network Monitoring," *Proceedings of the USENIX Winter 1992 Technical Conference*, pp. 333-343, San Francisco, CA, January 1992.

[2] Callaghan, Brent, Brian Pawlowski, and Peter Staubach, "NFS Version 3 Protocol Specification," http://www.ietf.org/rfc/rfc1813.txt, June, 1995.

[3] Combs, Gerald, *ethereal*, http://www.ethereal.com/.

[4] Curry, Dave and Jeff Mogul, *nfswatch*, http://freeware.sgi.com/cd-3/relnotes/nfswatch.html.

[5] Dahlin, Michael, Randolph Wang, Thomas E. Anderson, and David A. Patterson, "Cooperative Caching: Using Remote Client Memory to Improve File System Performance," *Proceedings of the First USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 267-280, Monterey, CA, 1994.

[6] Ellard, Daniel, Jonathan Ledlie, Pia Malkani, and Margo Seltzer, "Passive NFS Tracing of Email and Research Workloads," *Proceedings of the Second USENIX Conference on File and Storage Technologies (FAST'03)*, pages 203-216, San Francisco, CA, March, 2003.

[7] Ellard, Daniel, Jonathan Ledlie, and Margo Seltzer, "The Utility of File Names," *Technical Report TR-05-03*, Harvard University Division of Engineering and Applied Sciences, 2003.

[8] Elprin, Nicholas and Bryan Parno, "An Analysis of Database-Driven Mail Servers," *Proceedings of the 17th Large Installation Systems Administration Conference (LISA'03)*, San Diego, October, 2003.

[9] Hitz, D., J. Lau, and M. Malcolm, "File System Design for an NFS File Server Appliance," *Proceedings of the USENIX Winter 1994 Technical Conference*, pp. 235-246, San Francisco, CA, January, 1994.

[10] Jacobson, Van, Craig Leres, and Steven McCanne, *libpcap*, http://sourceforge.net/projects/libpcap/.

[11] Jacobson, Van, Craig Leres, and Steven McCanne, *tcpdump implementation*, http://www.tcpdump.org/.

[12] Moore, Andrew W., *Operating System and File System Monitoring: a Comparison of Passive Network Monitoring with Full Kernel Instrumentation Techniques*, Master's thesis, Monash University, 1995.

[13] *The NFSv4 project*, http://www.nfsv4.org/ .

[14] Nowicki, Bill, "NFS: Network File System Protocol Specification," http://www.ietf.org/rfc/rfc1094.txt , March 1989.

[15] Roselli, Drew, Jacob Lorch, and Thomas Anderson, "A Comparison of File System Workloads," *USENIX 2000 Technical Conference*, pp. 41-54, San Diego, CA, 2000.

[16] Shepler, Spencer, Carl Beame, Brent Callaghan, Mike Eisler, David Noveck, David Robinson, and Robert Thurlow, *The NFS Version 4 Protocol*, http://www.ietf.org/rfc/rcf3530.txt , May, 2000.

[17] Stern, Hal, Mike Eisler, and Ricardo Labiaga, *Managing NFS and NIS, Second Edition*, O'Reilly & Associates, 2001.

[18] Sun Microsystems, Inc., nfslogd(1M), *Solaris 8 Reference Manual Collection.*

[19] Sun Microsystems, Inc., snoop(1), *Solaris 8 Reference Manual Collection.*

[20] Williams, Thomas and Colin Kelley, *gnuplot*, http://www.gnuplot.info .

EasyVPN: IPsec Remote Access Made Easy

Mark C. Benvenuto and Angelos D. Keromytis – Columbia University

ABSTRACT

Telecommuting and access over a Wireless LAN require strong security at the network level. Although IPsec is well-suited for this task, it is difficult to configure and operate a large number of clients. To address this problem, we leverage the almost universal deployment and use of web browsers capable of SSL/TLS connections to web servers and the familiarity of users with such an interface. We use this mechanism to create configurations and certificates that will be downloaded to the user's machine and be used by a program to perform all configuration on the user's system.

Our system builds on common security protocols and standards such as IKE, X.509, and SSL/TLS to provide users with a secure-access environment that "just works." One of the main goals of the system is ease of use both for the users and the system administrators that maintain the infrastructure. We describe our implementation that uses Linux FreeS/WAN and Windows to show the practicality of the approach.

Introduction/Purpose

The growth of untrusted networks, e.g., the Internet and Wireless LANs, has created a problem for system administrators who need to provide secure access to corporate LANs for their users. While these problems superficially seem different, they can both be solved by using a virtual private network.

Native support for IPsec [12] in most operating systems allows most groups to create their own IPsec VPN solutions. Operating systems such as Windows 2000/XP, Linux, OpenBSD, and others support the ability to be IPsec clients and servers. While these systems are powerful and flexible, the configuration is often confusing and time-consuming. A tool that simplifies configuration is particularly important, because it allows system administrators to enforce policy in a transparent fashion, and it prevents user misconfiguration problems which are time consuming and expensive to fix.

The proposed framework is designed to eliminate those problems while remaining adaptable to the requirements of different sites. Because of the interoperability of IPsec, the server platform can be any platform, such as Linux or OpenBSD. The client platform chosen was Windows 2000/XP, because it has native IPsec support unlike Windows 9x/ME, and because it is the most common commodity OS platform. Our tool, like most VPN systems, models a remote access server (RAS) that is both simple and familiar to most users.

Background

Using IPsec for Remote Access is becoming important, but requires additional extensions and application support to become truly useful. IPsec is an appealing technology for VPN, remote access, and other uses because it is a proven standard that can be used as a trusted component in a larger security infrastructure.

The concluded IETF Working Group for IP Security Remote Access (IPSRA) [7] investigated the application of IPsec for Remote Access. This resulted in the PIC [14] proposal based on ISAKMP, and the (now expired) GetCert work [3]. The IPSRA working group also produced a set of requirements for IPsec Remote Access [16] to serve as guidelines for any standard. These guidelines divide remote access into a few categories, according to requirements such as location and level of security. This paper presents a system designed to fulfill the requirements for the "Telecommuters (Dialup/DSL/Cablemodem)" scenario.

The proposed GetCert uses a similar design to this paper, but it differs in that it uses a subset of the Simple Certificate Enrollment Protocol (SCEP) [13] as the certificate protocol which has support for root certificate, and CRL retrieval. Authentication is handled using a RADIUS server over a HTTP/TLS connection. This authentication scheme supports a variety of authentication techniques, but the protocol has no support for exchanging IPsec policy information with the client.

One early solution to the remote access problem was Moat [4], which was created by AT&T Laboratories and the FreeS/WAN project. Instead of requiring the client machine to run VPN software, Moat is a VPN/NAT appliance that contains an *x86* machine running Linux and FreeS/WAN designed to provide a dedicated VPN to a pre-configured corporate network. While it is a good solution for telecommuters who always work from the same place, it does not help in the "roaming salesman" access scenario. Also, because of its dedicated hardware and configuration, it is difficult to upgrade the software or use at other locations.

Another way to create IPsec tunnels is by using DNS KEY records such as the FreeS/WAN Opportunistic Encryption (OE). This system will initiate IPsec

connections with a potential host that has KEY and TXT records in its forward and reverse DNS zone files. This system makes creating tunnels easy as the system will make tunnels to any machine it can without any user involvement after the initial setup. The difficulty with this system is that it requires access to a forward domain to initiate connections, and the reverse DNS for it to be able to receive connections. These are steep initial setup requirements for inexperienced users.

Every time a connection is made to a new machine, the originator will have to do a reverse DNS lookup for the destination host to determine if a new connection can be made, a process that can be time consuming. Because of the need for frequent DNS lookups, the documentation recommends using a local caching DNS server for performance reasons. Finally, it is difficult for the user to determine which connections are protected without using FreeS/WAN status commands.

Wireless LANs (WLANs) have become increasingly popular in the last few years. Developed by the IEEE in standard 802.11, the WLANs have little security due to weaknesses in the wireless encryption [18], and other security features of the protocol. Because of the lack of security at the link-level, application and/or network level security protocols are used to address the problem, such as SSH, SSL/TLS, and IPsec. WLAN also lacks the benefits of some of the inherent physical security associated with wired networks since anyone in range of a wireless Access Point can potentially sniff network traffic or launch attacks.

One solution to the WLAN authentication problem was developed by the NASA Advanced Supercomputing Division. The "Wireless Firewall Gateway" (WFG) [1] serves the dual role of protecting the corporate LAN via a firewall and providing network configuration to clients via DHCP.

The WFG is an OpenBSD PC which runs a webserver that authenticates users over HTTPS against a RADIUS server with MD5 digest encryption, and then modifies firewall rules to allow client machines through the firewall. This machine relies on providing DHCP to the wireless clients so that it can be notified as clients connect and disconnect from the network. When a client releases or loses its DHCP lease, the WFG machine will remove the associated firewall rules. The system does not provide any additional protections to the clients because it is designed to protect access to the corporate network. WFG uses mutual authentication using an SSL certificate signed by VeriSign so clients can be reasonably assured that they are connecting to the server.

The power of this system is the hierarchy of security levels that are allowed. It is configured so that unauthenticated clients can be given limited access to only use email, VPN, and web access, while authenticated clients are given full access. The authentication relies on a modified ISC DHCPv3 server that will automatically update firewall rules as clients disconnect. One issue is that there is a small window of time between when a client that disconnects without properly notifying the DHCP server, and the DHCP server terminating the lease. In this window of time, it is possible for another client to spoof the identity of the recently disconnected user, and therefore gain authenticated access. Finally, this system relies on the user to properly use the network connection, which does not always happen as users often will use insecure protocols such as unencrypted email over the unsecured wireless network. This system is similar to the freely distributed NoCatAuth project from the NoCatNet project (http://nocat.net/).

WAVElan SECurity using IPsec (WAVEsec) (http://www.wavesec.org/) is a system designed to secure WLAN traffic by creating IPsec tunnels to a gateway on a local LAN. This system was designed for conferences to provide encrypted connections over WLAN to the public Internet. It does not authenticate users and requires custom software to configure connections. It provides initial IPsec connection information over DHCP packets such as the gateway IP. The server then sends its public key to the DHCP client as part of a dynamic DNS update request. Finally, the client creates the tunnel using the provided information, and the public key for the gateway from a DNS KEY record.

This system requires modifications to both the DHCP client and server to exchange configuration information and keys. It is designed to make it easy to create tunnels without exchanging keys or configuration out-of-band after the software is setup. It is designed to be used by experienced Linux users since the client setup requires installing custom scripts. Finally, this system is designed to only provide secure unauthenticated connections.

Problem

IPsec is a good solution for both remote access and wireless LAN access. Those two domains are similar enough that a common solution is feasible, without introducing undue complexity. This allows a system administrator to use a common set of software and hardware to provide IPsec for both sets of clients while being able to add additional security features as needed in the form of IPsec policies, firewalls, and DHCP servers, depending on the domain.

IPsec will only be practically useful to users if they can simply just use it without worrying about how it works. Ideally, users will be presented with some login screen where they will provide authentication information which is passed to an authentication server, and then the client machine will create a tunnel with the IPsec server. This ideal system will fit the two conflicting goals of providing a secure connection, yet be easy to use. The user should be confident that he is connecting to the correct server without danger of man-in-the-middle or impersonation attacks.

A current problem with IPsec is the many different implementations which all implement a different set of features for Internet Key Exchange (IKE) [11], and have distinctly different ways of configuring them. Implementations support some or all of the following authentication methods: private shared key, RSA public/private key, X.509, and manual keying. Many implement private key exchange, but this can be difficult to manage as the number of distinct users grows, difficult to work with in dynamic environments, and requires out-of-band cooperation from both client and server to setup. X.509 certificates solve many of these problems, but also require cooperation to configure.

The biggest problem in the configuration process is configuring the correct set of certificates to use, especially for someone who is not familiar with the details of X.509 certificates [2] and the intricacies of the different implementations. One of the authors spent many hours dealing with certificate problems in order to make an IPsec tunnel. There problems provided good evidence of the problems people may have and motivation to solve them. The complexity in the process can be eliminated with a client-side tool that will do all the certificate configuration and additional sanity checking as needed. On the server side, scripts can validate the setup with useful and instructive error messages about incorrect certificates without requiring that the system administrator become an IPsec expert to diagnose the logs. Such tools will only check each system in isolation for configuration problems but not large end-to-end problems such as firewalls.

To verify the security of the network configuration, the system administrator should use a tool such as IPSECvalidate [17] or other verification approaches [5]. Verifying the actual configuration is an important part of setting up an IPsec solution so users can correct security lapses, and it prevents unwanted access to the corporate network.

Other important features for system administrators include easy manageability, such as a system that will warn users about expiring tickets and performance metrics so usage growth can be anticipated. SNMP is the traditional mechanism used to monitor systems and MIBS for both IKE [6] and IPsec [8, 9] are in development.

Architecture

The system architecture, shown in Figure 1, is composed of three systems: client, gateway, and VPN Server. In our approach, we leverage the almost universal deployment and use of web browsers capable of SSL/TLS connections to web servers, and the familiarity of users with such an interface. We use this mechanism to create configurations and certificates that will be downloaded to the user's machine. This information is used by a program that will perform all IPsec configuration on the user's system using the server-provided configuration. The client and servers do all authentication via X.509 certificates in both IKE and SSL/TLS.

The *client* is the IPsec client that will create a certificate with the gateway, retrieve IPsec tunnel configuration information, and then create the IPsec tunnel with the VPN Server. The client program will generate a certificate signing request (CSR) and private key. The client chooses the gateway either by a hard-coded IP or hostname, or via an SRV DNS record for the destination domain. An SSL/TLS connection is then made to the gateway machine that receives the user's username, password, and CSR as part of a HTTP POST operation. The client then receives the signed certificate back in a HTTP response, with configuration information indicating various network parameters such as the tunnel destination IP address and network mask. The client is responsible for initializing the IPsec connection to the server with these network parameters and the freshly-minted certificate.

There are two important details. First, the SSL/TLS connection establishes the identity of the gateway server either via the system certificate store in the case of Windows or via an embedded certificate in the application. If a system certificate store is used, then either a third-party certificate authority such as VeriSign or a local certificate authority must sign the gateway certificate. In the second case, either the user must configure the new CA or the application can configure the CA for the user. If the application needs to configure the certificate, then the CA certificate needs to be embedded in the application, which requires the user to be able to initially obtain the client in a secure manner such as a floppy disk from friends or via a trusted network before trying remote access.

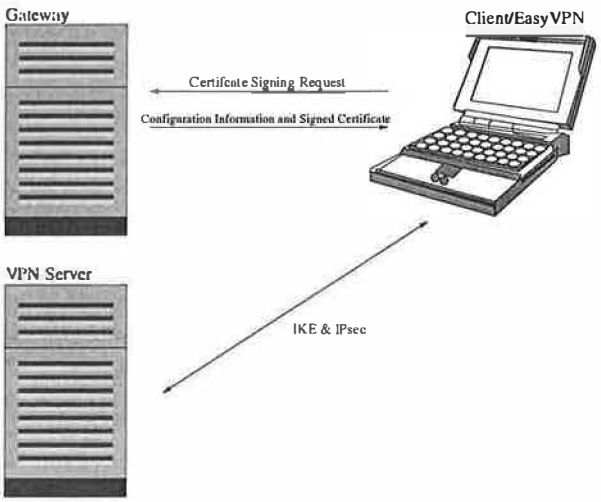


Figure 1: System architecture.

Second, the configuration information is extensible. Currently, it is a simple text format, but other formats can be used, such as XML. There are no restrictions on the content of the policy passed to the client.

Useful information could be preferred ciphers, a message of the day, or other additional parameters needed to configure the tunnel. This allows the protocol to adapt to the needs of the client/server configurations.

The *gateway* is running an HTTPS-enabled web server that serves as a Certificate Authority. Its task is to produce X.509 certificates for use in user authentication during IKE exchanges. The requests are received via HTTP POST requests, which are then processed by a CGI script to authenticate the user and sign the certificate after successful authentication. The user can be authenticated in any appropriate manner such as Unix PAM, or a RADIUS server.

The gateway also serves as a policy server such that it provides configuration information for the client to use to create the IPsec tunnel with the IPsec server. This allows it to dynamically load-balance clients between servers, favor certain users, or restrict users differently according to predefined classes such as user groups or location. While it is desirable to simply enforce these policies only on the client side, because of the open nature of the protocol, the policies need to be enforced on the server side in the form of firewall rules, additional X.509 certificate rules, etc.

The *VPN Server* is simply required to negotiate with clients that present valid certificates signed by the CA. Clients verify the server's authenticity via a certificate signed by the certificate authority. The VPN server also authenticates the client in the same manner. This means that the VPN server is not required to keep a list of clients or exchange any state explicitly with the gateway.

This architecture is very powerful because neither the gateway nor the VPN server need to keep state about the allowed users. The VPN server will accept users because their certificates are signed by the CA, and the client accepts the server because its certificate is signed by the same CA. The certificates can be used to implement policies that require state by treating it as a token. Since the certificate is passed to the server for new connections, digitally signed by the CA, and supports custom fields, it provides a secure extensible token to use to exchange simple policy information. For instance, a timeout for the IPsec connection can be implemented by using a short lifetime for the certificate to force the VPN server to deny the client connections after the timeout or attempts by the client to re-key the connection.

This stateless interaction between the gateway and the VPN server allows for multiple gateways and servers. Since the gateway is simply an HTTPS server, the servers can be load balanced with fail-over support using existing techniques to improve the reliability of the system. For the VPN servers, the CGI script used to generate the policy can choose a VPN server for the client to use from a collection of such servers. The criteria for choosing the VPN server to use may be round robin, smallest number of IPsec connections, network locality, etc.

Application Models

There are two models for building the client. The client can either be a separate application that resides on the client machine or be a browser-based plug-in that interacts with the system only when the user accesses a specific page. Each model has specific advantages and disadvantages, but only the application model is easily adaptable to multiple operating systems. A common problem to both models is developing new client software. Since IPsec configuration is platform specific, developers are still faced with creating different solutions for each platform. The IETF has started the process of developing an API to solve this problem [15].

Standalone Application

The stand-alone application is the original design of the application. The core requirements are to keep the binary size small and to minimize the installation complexity. In our implementation, the size was kept to a minimum by using only native Win32 API for all GUI, cryptography, certificate and HTTPS tasks, instead of external libraries or frameworks.

This approach has two advantages. First, the client resides on the client machine, making it easy for the user to access it without requiring a web browser. Second, the application model is a portable concept that can apply to other operating systems and user environments without forcing the user to use a particular web browser.

There are also two disadvantages. First, in order to build the first stage of trust, the client has to trust the CA certificate used by the system which either needs to be signed by a well-known CA so that the CA already exists in the system store or embedded so that the client application can create the trust itself. If an embedded certificate is used, then that means the client itself has to be trusted beforehand. Second, updating the client becomes difficult since this requires special client support to do automatic updates or user intervention to download a new client.

Browser Plug-in

The browser-based plug-in is an extension of the stand-alone application. It operates in a different way than the stand-alone application, but must meet the same requirements.

The most important requirement for a browser plug-in is what browser and operating system combination to use. Two of the more popular browsers are Microsoft Internet Explorer and Mozilla, which implement different approaches for supporting plug-ins, each with different limitations. Both support ActiveX components on Windows that can be downloaded from the network and installed by the browser. These ActiveX components have full access to the operating system services after the digital signature of the component is verified. On other platforms, such as Linux,

Mozilla is not a suitable platform for this task. The only dynamic plug-in support is through its JavaScript and XPCOM system. Unfortunately, the system requires lower level access to system functionality than provided by the Mozilla platform.

This approach has three advantages. First, updates are easy as the client accesses the web site to get the client each time. Second, the user is required to trust the web site he is accessing and this trust has been setup out-of-band via a third party CA or other arrangement. Third, the plug-in can be more adaptable by simply trusting the location of its download as a trusted source instead of requiring hard-coded configuration parameters.

There are also two disadvantages. First, we require the user to establish trust with the gateway web server, which can be difficult to do correctly if the user misconfigures their web browser to trust the wrong set of certificates. This flexibility may be too much for system administrators to give to users for fear of users incorrectly configuring their certificates, and therefore being open to attacks from imposter servers. Second, the flexibility required to support the plug-in approach is limited to browsers on Windows because no other web browser and operating system pair supports dynamic plug-ins that can access systems services. This is also difficult on multi-user operating systems because users usually need an elevated level of access to be able to manipulate the IPsec subsystem.

Implementation

The architecture is a framework in which any implementation can fit as long as it uses the correct set of protocols. For instance, there are several operating systems that support IPsec and IKE, and the choice may be made simply because of familiarity or availability. The client, on the other hand, depends on the individual user. For our implementation, we used Linux with FreeS/WAN on the server because of our familiarity with it and its availability. For the client, we chose Windows 2000/XP because of its support for IPsec and its popularity as a client platform. We believe that the ability to support such diverse platforms demonstrates the flexibility of the architecture.

Client

This implementation supports any *x86* system running Windows 2000 SP2 or Windows XP. These clients support IPsec tunneling natively as opposed to the Windows *9x* series. In order to configure the IPsec subsystem for Windows, the user must either manipulate it with dialog boxes and property pages via the Microsoft Management Console (MMC) or with the Internet Protocol Security Policies Tool (Ipsecpol.exe). There is no public API to access the IPsec subsystem and so the only practical way to configure IPsec is through the command line tool [10].

A unique advantage of using Windows as the client is the centralized Certificate Store. The certificate store can be accessed through MMC, and is used by both Internet Explorer and the IPsec subsystem. This makes it easy to manage certificates since there is one comprehensive place to establish and analyze trust relationships. This was useful while developing the client because it allowed flexibility in testing and made it easy to check what the system trusted. Once a certificate is placed on the certificate store, it can be accessed by the WinInet API, Internet Explorer, and the IPsec subsystem.

Server

The IPsec server can be any system that supports IPsec and has an IKE daemon capable of supporting X.509 authentication via a Certificate Authority certificate. Systems that support this include Linux/FreeS/WAN, OpenBSD, and {Net,Free}BSD with the Raccoon IKE engine.

For our implementation, a Linux *x86* Redhat 7.3 system with FreeS/WAN and the FreeS/WAN X.509 patch was used. To simplify the configuration, the gateway and VPN server ran on the same machine. For the gateway web server, we used Apache with the mod_ssl module. The Certificate Authority was a Perl script that manipulated the certificates using OpenSSL.

Security

The security of this system depends on its design, its implementation, and use. The design uses well-understood protocols as building blocks with a minimal trust relationship between the various machines comprising the framework. Each of the protocols has been evaluated for several years, so their weaknesses are well-understood. The implementation is built on proven system libraries. Overall, the system is secure when used by intelligent users.

SSLv3 and TLS have been resistant to many attacks and there are several implementations such as OpenSSL, Microsoft WinInet API, etc. Its security relies on the user being acquiring and using valid certificates.

X.509 has also not been successfully attacked, but its security relies on the ability to establish trust and the secrecy of the private keys. While it is recommended that the Certificate Authority be kept separate from the network, our system requires the CA to be connected. The security risks involved can be mitigated by keeping even the CA separate from the gateway, and by creating a separate certificate hierarchy to handle certificates for IPsec. For instance, the CA used to generate certificates for the IPsec gateway and server should not be used to sign the certificate for a company's main web site. Either a separate subtree or disjoint certification tree should be used, instead of the normal CA tree.

Finally, IPsec is composed of the IPsec protocol, and the key exchange. The IPsec protocol is built on simple symmetric cipher and hash operations. While its security is well understood, IKE is a complex protocol, whose security properties are still not entirely understood. This complexity also leads to large and

bug-prone software implementations, which introduce additional risks.

While the protocols can be shown to be secure against cryptanalytic attacks, the system is still vulnerable to problems created by careless users. For instance, by allowing users' computers access to the internal network, we could allow worms to spread, or increase the network's exposure to attack. Many windows viruses spread via HTTP (Code Red, Nimda) or SMB file shares (Klez, BugBear). Also, the new tunnel is another potential liability, since it presents another computer which an attacker can compromise and thereby access the corporate network.

Conclusions and Future Work

We presented a framework for providing strong security at the network layer to telecommuters and Wireless LAN users by using IPsec. Our contribution is in the ease of use for end users and administrators, by using familiar tools and interfaces (such as a web browser and SSL authentication), which we use to automatically configure the end-user's system with the appropriate parameters. Our system thus removes a large impediment to the use of secure protocols such as IPsec, which are otherwise difficult to configure. The proposed architecture can support any combination of server and client platforms, and can be tailored to the specific needs of individual sites and organizations.

There is more work to be done on implementing this system for other operating systems. Also, the protocol needs to be improved and formalized possibly by using a language like XML. Finally, the system would be integrated with a firewall and/or network appliance, to provide a turn-key solution for telecommuters and users of wireless LANs.

Acknowledgements

This work was supported in part by NSF under Contract CCR-TC-0208972.

Author Information

Mark Benvenuto received his B.S. in Computer Science from Columbia University (2003). While a student, he worked part-time as a junior system administrator for the Computer Science Department. He currently works as a Software Design Engineer in the SQL Server Engine group at Microsoft. He can be reached at markb@cs.columbia.edu .

Angelos Keromytis has been an Assistant Professor in the Computer Science Department at Columbia University since 2001. He received his M.Sc. and Ph.D. in Computer Science from the University of Pennsylvania (2001), and his B.Sc. from the University of Crete, Greece (1996). His research revolves around end-point security mechanisms, cryptographic protocols, and operating system support for security. He can be reached at angelos@cs.columbia.edu .

References

[1] Boscia, N. and D. Shaw "Wireless Firewall Gateway White Paper," NASA Advanced Supercomputing Division, White Paper, http://www.nas.nasa.gov/Groups/Networks/Projects/Wireless/index. html , March, 2003.

[2] CCITT, *X.509: The Directory Authentication Framework,* International Telecommunications Union, 1989.

[3] "Client Certificate and Key Retrieval for IKE," IETF draft, work in progress, http://www.ietf.org/proceedings/01mar/I-D/ipsra-getcert-00.txt .

[4] Denker, John S., Steven M. Bellovin, Hugh Daniel, Nancy L. Mintz, Tom Killian, and Mark Plotnick, "Moat: a Virtual Private Network Appliance and Services Platform," *Proceedings of LISA*, pp. 251-260, 1999.

[5] Fu, Z., S. F. Wu, H. Huang, K. Loh, and F. Gong, "IPsec/VPN Security Policy: Correctness, Conflict Detection and Resolution," *IEEE Policy 2001 Workshop*, Jan., 2001.

[6] "Internet Key Exchange (IKE) Monitoring MIB," IETF draft, work in progress, http://www.ietf.org/internet-drafts/draft-ietf-ipsec-ike-monitor-mib-04.txt .

[7] IP Security Remote Access (ipsra) Working Group, http://www.ietf.org/html.charters/OLD/ipsra-charter.html .

[8] "IPsec Flow Monitoring MIB," IETF draft, work in progress, http://www.ietf.org/internet-drafts/draft-ietf-ipsec-flow-monitoring-mib-02.txt .

[9] "IPsec Monitoring MIB," IETF draft, work in progress, http://www.ietf.org/internet-drafts/draft-ietf-ipsec-monitor-mib-06.txt .

[10] "KB265112: IPsec and L2TP Implementation in Windows 2000," Microsoft Knowledge Base, June, 2003.

[11] Kent, S. and R. Atkinson, "The Internet Key Exchange (IKE)." RFC (Proposed Standard) 2409, IETF, November, 1998.

[12] Kent, S. and R. Atkinson, "Security Architecture for the Internet Protocol," RFC (Proposed Standard) 2401, IETF, November, 1998.

[13] Liu, X., C. Madsen, D. McDrew, and A. Nourse, "Cisco Systems' Simple Certificate Enrollment Protocol (SCEP)," IETF draft, work in progress, http://www.ietf.org/internet-drafts/draft-nourse-scep-06.txt , May, 2002.

[14] "PIC, A Pre-IKE Credential Provisioning Protocol," IETF draft, work in progress, http://www.ietf.org/internet-drafts/draft-ietf-ipsra-pic-06.txt .

[15] "Requirements for an IPsec API," IETF draft, work work in progress, http://www.ietf.org/internet-drafts/ draft-ietf-ipsp-ipsec-apireq-00.txt .

[16] "RFC 3457: Requirements for IPsec Remote Access Scenarios," http://www.ietf.org/rfc/rfc3457.txt .

[17] Sailer, R., et al., "IPSECvalidate – A Tool to Validate IPsec Configurations," *Proceedings of LISA*, 2001.

[18] Stubblefield, Adam, John Ioannidis, and Aviel D. Rubin. "Using the Fluhrer, Mantin, and Shamir Attack to Break WEP," *Network and Distributed System Security Symposium Conference Proceedings*, 2002.

The Yearly Review, or How to Evaluate Your Sys Admin

Carrie Gates and Jason Rouse – Dalhousie University

ABSTRACT

While some work has discussed hiring system administrators, and other work has focused on the technical and mechanical requirements for terminating a system administrator, there has been very little published regarding how to review or evaluate a system administrator. This paper presents one approach to doing such a review, followed by scenarios that explore the approach. The system developed in this paper has the aim of creating measurable goals that a competent system administrator should be able to achieve. We also discuss when the use of this model is appropriate, its strengths and weaknesses, and the responsibilities placed on management if this model is used.

Introduction

There are several publications that talk about how to hire a system administrator (e.g., [5, 6]). In these cases, the emphasis is on how to determine the applicant's problem solving ability, general knowledge, and fit within a company. There are fewer publications dealing with how to fire a system administrator (e.g., [5, 7]). Those publications that do discuss how to fire a system administrator concentrate on the technical aspects: how do you ensure that the system administrator no longer has system access, and that there are no backdoors, for example.

However, there is no previous work that discusses how to evaluate the effectiveness of a system administrator. The learning of system administration seems to be based largely on the apprentice system, with the implicit assumption that junior system administrators will learn from senior system administrators. It is further assumed that senior system administrators, by virtue of years of experience, are competent. But what if the junior administrators are taught poor practices? Unfortunately, it is not always the case that the senior administrator is competent, regardless of years of experience. Nor is it always the case that the system manager is himself an administrator, and so knowledgeable of the area and able to teach or evaluate his staff. Thus, guidelines need to be developed to assist in evaluating the effectiveness of system administrators.

This paper presents one approach to the evaluation of system administration personnel. We believe that there are three main criteria against which system administrators can be evaluated: achievement of goals (e.g., installing and deploying a back-up system), achievement of a specified service level (e.g., that the time between a user request and fulfillment of that request is less than some specified value), and general competence.

The first criterion focuses on the development of work-related goals that are mutually acceptable to the administrator and manager, where performance is later measured against the achievement of these stated goals. This emphasizes coordination and cooperation between administrator and manager, and is geared towards allowing non-technical managers to understand and evaluate administrator performance in an objective manner.

The second criterion concentrates on meeting "standard" service levels. Service levels include minimizing unscheduled downtime and having a guaranteed response time for users. In this section we define four main components – availability, usability, security and customer services – and outline suggested service levels and measurements for each one.

The third criterion deals with the problem solving and general competence of the system administrator. We believe that the best form of system administration is to solve problems correctly the first time, rather than continuously "hacking" a system, as this latter approach leads to later problems with various services. It also compounds the complexity of fire-fighting and can make third party trouble shooting nearly impossible. Thus, the third portion deals with measuring how often an administrator fixes the same problem.

It should be noted that the ultimate goal of this paper is to start discussion within the community on how to evaluate system administrators. To stimulate this discussion, this paper presents some guidelines in the formation of an evaluation system, as well as a proposed system that as much as possible tries to quantify the art of system administration.

Criteria

Before addressing the specifics of system administration, it is appropriate to consider performance appraisal in general. While it is often the case that a manager, who may or may not be familiar with the details of an employee's job, must evaluate that employee, this process *should* be one that involves both manager and employee. It is important that evaluations be performed in an objective manner that evaluates the work performed, and the quality and outcomes

of that work, and not the personality of the employee or the personal biases of the manager. In addition to avoiding potential legal problems from a subjective review, this also provides the best approach to a review that is *fair*.

Approach

The ultimate goal in the design of this evaluation procedure is fairness, to the employee, the manager, and the organization. We feel that the best way to attain this goal is to evaluate strictly on the work performed, and not on any of the more subjective criteria (e.g., how well the employee gets along with others, whether the employee is cooperative, and the like).

How can a system administrator's work be evaluated? System administration can often seem like more of a black art than a definable job. And no two days in an administrator's life might ever be the same, but instead consist of multiple tasks that are often unpredictable, making concrete evaluation trickier. Perhaps the most important skill of any system administrator is problem solving, yet this skill also comes from experience: experience with similar problems, with similar software, and with the organization of the system in question. There are, sadly, no standardized or widely accepted tests available that will grade an employee's problem solving skills, let alone these skills in relation to system administration. Further complicating the matter is the distinct possibility that the manager evaluating the system administrator may not be familiar with the systems for which the administrator is responsible (e.g., the manager might have a Windows NT background while the àdministrator works with Solaris and AIX), nor is there any guarantee that the manager has ever been a system administrator.

To address these issues, this paper adopts a three-part approach that is centered around the ideal of administrators and managers working together. The first part adopts the concept of goals that are developed by both the administrator and manager. In this case, the administrator is measured by his progress toward goals he helped to set. The second part presents four key components of system administration work and suggests relevant measurements within these components. The third part attempts to measure how effective a system administrator is by measuring how much time he spends revisiting problems.

It must be stressed at this point that the evaluation procedure requires communication between the manager and system administrator. A manager cannot simply tell an administrator that they are about to be evaluated without having previously explained the criteria they are expected to meet. Similarly, a manager should not be kept in the dark regarding current or potential problems on which the system administrator is working.

Goals

The first part of the three-part approach presented here draws heavily on the suggestions made by King [2]. King defines "performance plans" where the manager and employee work together to plan for the coming year and to define what needs to be achieved. King notes that performance plans require five characteristics in order to be considered achievable: specific, measurable, time limited, realistic, and challenging.

That is, the plan must be *specific* so that both the manager and employee are clear on what is expected. For example, "keep servers up" is too vague, whereas "ensure that web site X has no more than two hours of downtime in the coming year" is more specific, and therefore *measurable*.

Time limited ensures that the employee and manager are aware of any deadlines. For example, a goal of "install a new back-up system" might be given lower priority by an administrator if the current system is still in place and working, whereas the manager might expect that the new system is online within a month and have promised as much to clients. Thus, *time limits* are required on all aspects of a performance plan.

Finally, the plan must be both *realistic* and *challenging*. A realistic plan should be obvious; it is unfair to expect an employee to complete tasks that are not possible or are unnecessarily stressful (e.g., install and configure the new back-up system, and deploy to 150 clients, by the next day). Challenging may be a little less obvious. King argues that employees should be challenged in their jobs so that they remain interested in their work and are given the chance to excel. Thus, any performance plan drafted by the manager and employee should provide this opportunity.

In this article, we use the term *goals*, which we define as having an equivalent meaning to King's definition of a performance plan. More recently, goals have received attention in the human resources literature, where they have received the moniker of "SMART" goals: Specific, Measurable, Action-Oriented or Attainable or Aggressive, Realistic, Time-Constrained or Tangible [8, 1, 9]. In this case "challenging" has been replaced with "action-oriented," where it is expected that the goal requires that the individual must perform some action in order to attain the goal, rather than to define a goal and then not have any means for pursuing it.

In our evaluation procedures, we expect the manager and administrator to work together to define SMART goals. Due to the constantly changing nature of most system administration environments, it is suggested that this meeting take place multiple times per year, perhaps every three or four months, depending on the natural cycles of the environment in question. This allows for a complete view of the environment and the evolution of facilities or capacity planning, as well as enabling administrators and managers to note when goals begin to move "off the rail."

By having the manager and administrator work together to set goals, we allow the manager to ensure

that the overriding client[1] requirements are understood by the administrator (e.g., two-hour response time to client problems). It provides the manager with a chance to ensure that the goals the organization considers important are being met. Conversely, it allows the administrator to ensure that the goals are not unrealistic (e.g., 100% uptime for the next two years). The administrator also has the opportunity to make the manager aware of any issues that might otherwise be missed (e.g., a major server is starting to have hardware problems and will need to be replaced soon).

In addition to providing input to the manager that will enable him to make more effective decisions, and providing the administrator with a sense of the larger picture in terms of corporate or organization goals, having the manager and administrator work together to determine objective goals will make the evaluation process consistent and fair. For example, a manager can not simply evaluate an administrator using the criteria outlined here, or any other criteria, without informing the employee of the criteria against which they will be measured. By working with the manager to create goals, not only is the employee aware of the criteria against which he will be measured, but he is also ensured a priori that both of them agree that these criteria are reasonable.

The authors recognize that there are three types of goals: personal, professional and organizational. Personal goals involve those goals a person might have that are in no way related to his profession (e.g., to become a better painter). Professional goals encompass the development of skills which are not necessarily directly related to the organization (e.g., to learn Perl by the end of the year) but may be used to the benefit of the organization. Organizational goals are those that are directly related to what the organization requires from the individual.

While all of the goal examples previously described were organizational in nature, it is also appropriate to include professional goals in the goal-setting section of the performance review. This gives the administrator the chance to learn new skills that, while not necessarily directly related to the immediate organizational goals, will indirectly benefit the company by allowing the administrator to remain current, and by keeping the administrator happy with the company.

Service Levels

The first stage in designing evaluation criteria for a system administrator is to identify the critical components of system administration. Here, we define these general components to be availability (of hardware, software, services and data, including backups), usability (whether users are able to perform the tasks they need to complete), security, and customer service. It should be noted that, ultimately, system administration is a service, and as much as we complain about "lusers," we are ultimately responsible to our users.

Within these four components, outcomes need to be defined. These outcomes need to be easily quantifiable and measurable, without necessarily requiring the services of an expert system administrator. The outcomes should be easy to gather, and should follow the KISS (Keep It Simple, Stupid) principle. Further, the outcomes should also build in a "benefit of the doubt," so that it can be recognized that the administrator made a legitimate mistake or that there were extenuating circumstances. In particular, junior administrators should be given more latitude than senior administrators.

The following questions in each of the four components have been identified as meeting the design criteria:

1. Availability:
   a. How often has the system, including hardware and key services such as web or ssh, had unscheduled downtime within the past year?
   b. Can a file that was deleted yesterday be reliably recovered from backup? Deleted last week? Last month? From both servers and desktops?
   c. How many mistakes has the system administrator made that directly led to system or service downtime?
2. Usability
   a. What is the average time between a user request being made and the fulfillment of that request?
   b. What is the average time to install a new server and have it operational?
   c. What is the average time to install a new service and have it operational?
   d. Identify the top 10 software and services used on a system. How far out of date is this software? Is there a legitimate reason for using old software (e.g., gcc 2.95 still required over 3.0)?
3. Security
   a. What is the average time between a relevant security patch being released and being installed?
   b. What is the average time between a problem occurring with the system and the administrator noticing? (Underlying question: Does the administrator monitor the system?)
   c. Are appropriate security measures installed and monitored? For example, is there an intrusion detection system, and are alerts monitored?)
   d. Is confidential material treated appropriately?
4. Customer Service
   a. What is the average response time when a user emails a system administrator?

[1] We do not want to constrain this to a business environment; academic and government facilities have clients too! We use the term client in a general sense, encompassing users and lower-level management, as well as the traditional business meaning.

b. Does the administrator notify users of events such as expected downtime or policy changes?
c. Does the administrator make an effort to stay current, either through reading appropriate mailing lists, taking training courses, reading books and magazines, etc.?
d. Does the administrator follow a practice of both learning from and teaching co-workers? (Or, does he have job security through obscurity?)

In some of these cases, reasonable values need to be determined, and may be dependent on the organization. For example, the ideal time between a relevant security patch being released and being installed should be relatively short (e.g., perhaps three days). The average response time to user email might be organization dependent, where some organizations expect a one business day turnaround, and others might only require a one week turnaround.

Competency

This section was the cause of much discussion among the authors and other system administrators. How does one evaluate the competency of system administrators? At the very least, how can one differentiate between a competent system administrator and one who may need to be appropriately trained and/or disciplined, or even replaced. This is not easy. For example, we have seen systems where there were five backup copies of the passwd file going back two years in /etc, along with a directory /etc/passwd.backup containing more backups of the password file. This was on a server that was also being backed up by two different systems: Amanda and TSM. How does one quantify system administration practices in such a manner that recognizes these practices as undesirable and unnecessary?

The one consistency that the authors could find is that poor system administration practices lead to the same problem being revisited multiple times. That is, if the system or service was installed correctly the first time, there should be minimal problems reported with the service. This is not to say that there will be no user requests. Rather, the user requests should take the form of "please install..." rather than "please fix..."!

Therefore, in order to provide some measure of general competency of the system administrator, the manager will need to follow user requests and track when those requests are due to a piece of software that was incorrectly installed or configured.

Measurement

As there are three parts to the evaluation, there are also three parts to the measurement section. The first two sections are worth 10 points each, while the third section is worth 5 points, for a total of 25 overall. The lower the overall score, the better the performance of the administrator. In the first two sections, 0 points implies that the standard (either goals or service level) was met. As it is possible to exceed the expectations for goals, it is possible to score lower than 0 points in this section. The third section is only worth 5 points. This is NOT an indication of its importance relative to the other two measures. Rather, it is a recognition of

Service	Response	Value Range
1. Availability		
(a)	Count the number of non-hardware related down-times (Hardware and power failures should not be counted here)	0 - 8
(b)	Count the number of no responses (out of 6)	0 - 6
(c)	Count the number of known occasions (Note that the manager might not know these!)	0 - 6
2. Usability		
(a)	Count the number of days	0 - 5
(b)	Count the number of days	0 - 5
(c)	Count the number of days	0 - 5
(d)	Add 0.5 for every unjustified full release difference Add 0.1 for every point release difference	0 - 5
3. Security		
(a)	Count the number of days	0 - 5
(b)	Count the number of hours	0 - 8
(c)	If no, add two points, else add zero	0 - 2
(d)	If no, add five points, else add zero	0 - 5
4. Customer Service		
(a)	Count the number of days	0 - 8
(b)	If no, add five points, else add zero	0 - 5
(c)	If no, add five points, else add zero	0 - 5
(d)	If no, add two points, else add zero	0 - 2

Table 1: Suggested measurements for the service levels.

the difficulty of measuring a system administrator's competence, and of the volatility involved in the process and subsequent discussions.

Measuring Goals

The first part, measuring the achievement of the goals set throughout the year, is the easiest. Given that there are performance evaluations annually, the measurement consists of "met some goals" (10 points), "met most goals" (5 points), "met all goals" (0 points), "exceeded some goals" (-2 points) and "exceeded most goals" (-5 points). As the process of setting goals requires the administrator and manager to meet periodically through-out the year, it allows time to adjust deadlines if necessary. For example, a goal might have been to install a new server by a particular date. However, if the server arrived one month later than expected, this should be factored into the goal deadlines.

Measuring Service Levels

Table 1 provides a suggested measurement scheme for part two of the evaluation. It should be noted that this is a suggestion only, and will need to be adjusted appropriately for each organization. For example, condition 4a, the average response time when a user emails an administrator, might more appropriately be measured in hours rather than days depending upon the environment. Or condition 2b, the average time to install a new server and have it operational, might better be measured in weeks rather than days, again depending on the complexity of the environment. The table presented provides suggested measurements based on a medium-sized faculty in a research-based university. Most organizations will likely want more strict values.

The measurements for this section result in a measure out of 80, so will need to be divided by 8 before being added to the total. Each of the four components is weighted equally in this scheme. However, some organizations might place a higher premium on some components over others (e.g., security over customer service). In these cases they should adjust the weights accordingly.

Some of these questions require further explanation at this point. First, it is assumed that a tracking system is in place so that the manager is aware of items such as the time between a request being made and the fulfillment of that request. If no such system is in place, then the manager requires a more hands-on involvement in the system administration of the organization in order to be able to answer some of the questions. In any case where the manager is not aware of the answer, the benefit of the doubt should *always* be given, and so a value of 0 should be assigned. The manager should never assign a higher value without the documentation to support his evaluation. Otherwise, the manager is no longer evaluating the work performed, but rather his personal beliefs of the worker's performance.

In the same vein, items such as 3d, is confidential material treated appropriately, should not be based on "gut feeling." Rather, a non-zero value should only be assigned if the administrator has actually performed some action that violates a user's confidentiality (e.g., printing credit card information and then leaving it in a public recycling bin rather than shredding it). Similarly, items 4b, email of notification of downtime or policy change, 4c, personal improvement, and 4d, collaboration with peers, should receive a value of 0 unless the manager has documentation otherwise. Documentation in this case might include a complaint from a user of not being informed of a system change (4b), or written confirmation from the administrator that he does not follow appropriate mailing lists or magazines, etc., (4c), or complaints from coworkers that the administrator does not share information on or help with systems changes (4d).

Measuring Competency

For the third section of the evaluation, the manager needs to review the system administration tasks for the period being reviewed (this should ideally be performed every time the manager meets with the administrator regarding the goals for the period, rather than only once per year), and compare these against the user requests made, as well as the jobs put in the job tracking system. The number of times that a user has repeated the same request before final resolution should be counted. For example, a user might request a Perl module be installed, which, once installed, might be followed by a comment that the module still does not work. This would count as one, as the user needed to repeat the request once before the service was usable to the user. Similarly, if a backup system was installed, and the task was set as completed, yet job tickets followed stating that machines X and Y were not backing up properly, this would count as two: one for machine X and one for machine Y.

It is recognized that administrators will make mistakes. It is further recognized that junior administrators will make more mistakes than senior administrators. However, mistakes should be minimal as administrators should test their installations, configurations and changes before checking the job as completed in the job tracking system. The manager should determine what he feels is an acceptable level of repeat incidents.

A suggested guideline might be that incidents such as those described above should happen no more than once per month on average for senior system administrators. Junior administrators should be given more latitude, allowed to make three mistakes per month on average. Therefore the administrator receives a value of zero for this section if mistakes were made no more often than listed for his level. For each additional month of mistakes (that is, assuming a junior administrator, for every three additional mistakes per year above and beyond the 36 allowed) one additional point is assigned, up to a maximum of five points.

Usage

The score achieved on each of these three sections can be added together to provide an overall view of the effectiveness of the system administrator, where lower numbers indicate better performance. While it is tempting to provide absolute values that categorize an administrator as either good or bad, the authors feel that the manager has the responsibility of determining what are acceptable values for each of the three portions of the evaluation, as well as the overall value. The manager is also responsible for communicating this to the administrator *before* the evaluation.

The results of the overall evaluation can be used to assist the manager in determining how best to use the administrator. For example, the administrator might perform strongly everywhere except for customer service related goals and service levels. In this case, the manager might move the administrator to a less visible administrative role and have another administrator be the primary contact for users.

Similarly, the manager might also be able to use the results of the evaluation to determine what training courses might be appropriate. For example, if the system administrator consistently misses some goals, then the manager should look for some common element in the goals missed. Were they all related to AIX systems? Or were they all related to Solstice Backup Server? By searching for the common element, the manager can recognize where particular training might be beneficial. A careful review of where a system administrator makes mistakes (as noted in the third part of the evaluation) can also lead the manager to determine where further training might be appropriate.

The results from the evaluation might also show that the administrator consistently exceeds the majority of the goals set, but might not perform so well under the service level section. In this case it might simply be a matter of structuring goals for the administrator so that the service levels are also achieved. Or perhaps the administrator was not aware of the service levels as separate goals that needed to be achieved, and so did not prioritize appropriately.

Finally, this evaluation tool can be used to determine the administrator's strengths and weaknesses. This allows both the manager and administrator to work on the weakness, and to take advantage of the strengths.

Scenarios

This section describes five different scenarios, with some typical measurements, and the responses to the evaluations. It is provided to give the reader a sense of how this process can be deployed, the types of responses it might generate, and how the administrator and manager can work together to solve issues.

The Happy Helper

The first case involves an administrator in a small university, working with a team of administrators. The university provides a number of servers and services to its students, faculty and staff.

The established goals centered around technical requirements, such as deploying a new printing system. At the end of the evaluation period, the administrator had met all goals, receiving 0 points.

For the availability and usability sections of the service levels evaluation, the administrator scored well, receiving 0 points for each. However, in the security section the administrator received 1 point for the time delay between a security patch being released and being installed, another point for the time between a problem occurring with the system and the administrator noticing, and 2 points for not having additional security measures installed and monitored. Under the customer service section, the administrator received 4 points for the response time to user email and another 2 points for not always notifying users of changes. The total number of points awarded was 10, resulting in 1.2 points (out of 10) for this service levels section.

For the third section, the administrator, who was expected to perform at the level of a senior administrator, had acquired 3 points (having, throughout the year, made 15 errors that resulted in services needing to be reconfigured). This resulted in an overall evaluation score of 4.2 (out of 25).

During the performance review, the administrator complained that he did not have time to monitor the system or respond to email because he was so busy helping the people who dropped by his office. Similarly, services were often deployed without complete testing due to these interruptions. He felt that his job included helping people, and so other areas suffered, and yet there was no recognition in the process of this service. In this case, the manager recognized that the administrator was correct (having often walked by his office and seen people in it!).

The manager agreed that helping people was part of the administrator's responsibilities. However, he felt that many of the questions could be handled by other personnel, freeing the administrator to perform more administration duties. A compromise was agreed to where the administrator would hold office hours, during which time he was available to others for consultation, while the manager would email everyone to inform them of this change.

The goals for the administrator were centered around eliminating the distractions during non-office hours by asking people to enter their problems into a ticket system or to come back during office hours, and to close his door and put down the blinds during non-office hours. Other goals related to improving user response time and service. While security was also identified as important, the manager did not want to overload the administrator with goals, thereby decreasing his chances of meeting them, so it was agreed that the manager would ask another administrator to take over the

security duties. The last goal set was that the administrator was to reduce the number of errors per year to an acceptable level. It was felt that this goal was complimentary to the first, since the number of distractions was a significant contributor to the number of mistakes made.

Mr. Job Security

The second case involves a small, for profit, Internet-based software company with one administrator. The company relies on a number of NT servers to provide a service to their customers, and so requires a high level of uptime (99.999%).

Before implementing the performance evaluations described, the manager felt that the administrator was doing an adequate job. However, he did not like the administrator's attitude. The administrator felt that he had job security, and so would not deal with user requests in a timely fashion.

For the first section of the performance review, the goals that were set for the administrator centered around ensuring service for the clients, such as bringing new servers online, and developing specifications for new servers that would meet demands, such as hot failover. The administrator met all goals set, and so received 0 for this part of the evaluation. Similarly, for the third section of the performance review, the administrator did not often revisit the same problem, and so received 0 points.

The second part of the evaluation centered around the service levels. In the section on availability, the administrator performed well, although it was noted that the backups were not always reliable, scoring 2 points here. For the usability section, the administrator also performed well, with the average time for installing servers and services being acceptable. It was noted, however, that the usual time between a user request and fulfillment of that request was three days, and that some of the software was out-of-date, resulting in a score of 4 points. For the security section, it was noted that while patches were applied on time, and there had been no issues with confidential material, there was also no initiative on the administrator's part to improve security, and so there were no intrusion detection systems, etc., resulting in another score of 2 points. Finally, on the customer service side, the average response time for email was two days and the administrator did not make an effort to stay current, resulting in 4 points. The final point regarding the administrator both learning and teaching co-workers was deemed irrelevant and so was ignored.

The final score for the service levels evaluation was 12 out of 80 (which reduced to 1.5 out of 10, resulting in an overall evaluation score of 1.5 out of 25), which was felt by the manager to be unacceptable. The manager sat with the administrator and discussed how most of the points dealt with user-related issues and general response time. The result was goals for the following evaluation period being set that centered around user response time. It was made clear to the administrator that he was expected to achieve these goals, or else go through a discipline process.

In this case, the manager was able to articulate in an objective manner the requirements for the administrator, without resorting to statements such as "I don't like your attitude." The administrator became aware of the importance placed on timely responses to users, and was given the chance to correct his behavior.

The following year, the performance review had much the same results. There was no noticeable improvement in the response to users, and so no change to the service levels review, receiving a score of 1.5 (out of 10) again. As in the previous year, no noticeable mistakes were made, and so a value of 0 was assigned. However, the goals section had been changed from the previous year to include a number of goals ensuring improved response to user requests. As these goals were not met, a score of 10 (out of 10) was received for this section. The overall evaluation was therefore 11.5 (out of 25), as compared to 1.5 the year before. This provided the management with documented grounds for dismissal, and so the administrator was fired.

Just Plain Overworked

The next case involves a system administrator, working as part of a team, in a government office. The administrators were responsible for a large server farm, as well as user workstations and laptops, for a very large and demanding user group.

During the performance evaluation, the administrator performed well in the third section, having made few mistakes of note. For the first section, the administrator had completed all goals. However, he had not met all of them on time, having missed some of the less critical goals by a few days.

For the service levels section, the administrator scored well on availability and security, with zero points for each. However, for usability, the administrator received two points for the time required to respond to a user request, five points for the time required to install a new server, and three points for the time required to install a new service. In the customer service category, the administrator received a further two points for the delay in responding to user email.

With a total of 12 points, the manager felt that the service level score was too high, and also found it disturbing that some of the goals were missed. In general, it was felt that this administrator, along with the others on the team, was very good.

Similarly, other members of the team received scores with deficits in both the customer service and usability categories, and some of them had also failed to meet some goals on time. This alerted the manager to the fact that there was a more general problem with the system administration team. Although the team members had individually reported problems with

overallocation of work, the manager had believed that the workload was appropriate. With these new measures, however, the team was able to establish a reasonable benchmark for their output. The manager was able to identify the need for a larger team. Along with individual reports and these measures, the manager presented a balanced case to his superiors for the addition of another administrator.

After careful evaluation, another team member was hired, and, during the next review period, customer service and usability scores improved greatly, while the achievement of goals by individual team members reached 100%.

The No-Win

The fourth case involves a system administrator for a large telecommunications company. The administrator was responsible for the testing and deployment of new applications for the Internet service provision division of the company.

At the beginning of each year, goals for the entire year were established. Rather than individualizing each set of goals, and meeting periodically to review them, overarching goals of the company were used, and the administrator needed to choose one or more of these pre-defined goals. For example, one goal might be to save the company money. The onus was then on the administrator at the end of the year to list the projects worked on, and how he had saved the company money on those projects.

The manager maintained a hands-off approach to the administration of the systems, and so was unable to comment on many parts of the service levels section of the performance review. Additionally, as the administrator was responsible for the testing and integration of new applications, much of the service levels section was not applicable. He therefore received a score of 0 for this section. Similarly, using the method described here, he received a score of 0 on the competency section.

While the manager recognized that large portions of the current review system were not applicable, no effort was made to modify the review to better reflect the administrator's responsibilities.

Additionally, the manager would often not inform administrators of upcoming projects, preventing them from preparing. As a result, much of the testing and integration was performed with little notice and tight deadlines. While not listed as part of the goals section, the manager would still note any time one of these deadlines slipped, and add it to the administrator's file.

As a result, the administrator's reviews would result in an overall score of 0, indicating good performance. However, his reviews would still list missed deadlines, with no explanation as to why these deadlines were missed included, nor with any admittance on the manager's part that better communication would solve many of these issues. The administrator was put in a defensive position where he needed to indicate how his performance had been outstanding, yet not measured by the current system. As a result, the evaluation system being used was essentially ignored, and provided little guidance or feedback to either administrator or manager on how to improve the company's environment.

Awesome Admin

The final case involves a system administrator in another small, Internet-based company. However, in this company, there are multiple system administrators, each responsible for different portions of the system (that is, the database administrator is separate from the Unix administrator, is separate from the web administrator, etc.).

The goals for the Unix administrator again centered around technical accomplishments, such as installing a new backup system. The administrator easily met all goals set, receiving a score of 0. Similarly, he made very few mistakes, and so received a score of 0 for the third part of the evaluation. For the service levels section, the administrator again scored 0, having met all of the expectations set forth. The total score for the entire evaluation was therefore 0.

In this case, the manager sat with the administrator to determine how to better challenge the administrator, and to determine areas that could be improved. The administrator noted a desire to learn more about security, and so goals were set that included training and obtaining appropriate certification. Additionally, the administrator was given the task of performing a security audit of the entire company, intended to both allow the administrator to learn as well as provide the company with valuable feedback on how it could improve its security processes. As such, the goals shifted from technical accomplishments directly related to the job to be performed (organizational goals), to professional goals that also benefited the company.

Recommendations

Using this approach implies a certain level of maturity in both the people and the processes. The manager must be willing to work with the administrator, particularly in terms of setting appropriate goals. This approach is not intended to be sprung on an unsuspecting administrator as a means of termination, but rather to allow a manager to identify areas needing improvement and to determine how best to address these short-comings. Similarly, it allows an administrator to show how good his performance is (self-promotion of system administrators can't hurt!) to a manager who might not otherwise understand what the administrator does.

Given this, the authors have two recommendations for both managers and administrators. First, always treat each other with respect. This promotes

more open communication. For example, the manager should not talk about how the employee needs to improve, but rather about how the work needs to improve (e.g., more testing should be performed before stating that a task has been completed). This helps to take what can become a very antagonistic situation and keep it focused and non-personal.

Secondly, get everything in writing and signed. Once goals are agreed to by both parties, they need to be put in writing, and both the administrator and the manager need to sign that this is what they have agreed to. Both the manager and administrator should receive copies of this form. This serves to protect both sides: the administrator in the event the manager claims goals that had not been stated, and the manager in the event that the administrator states that he was never told that he needed to perform a particular duty.

Finally, a word of advice to managers in particular: talk to your human resources department! If you are in a large organization, it is likely that they already have recommendations, and possibly even requirements, on how to perform evaluations, as well as being able to provide advice on how to give an appraisal interview (see also [3] for advice on this) and how to handle difficult situations, such as the "poor performer."

Disclaimer

It should be noted that this approach, while intended to be general, will not necessarily be appropriate for all situations. For example, it assumes that there is already a job tracking system in place, which is an indication of an environment that is following a more mature process, rather than, for example, a small company struggling to get started. (See Kubicki [4] for a good description of mature system administration processes.)

There was also some discussion on where it would be appropriate to deploy this process. In the situations where there is a good system administrator and a good manager, it was felt that this approach would likely only formalize the arrangements that were already likely in place informally. In the cases of a poor administrator and a good manager, this approach provides an opportunity for the manager to work with the administrator to address his shortcomings and help with prioritization. In those cases where the administrator is unable to fulfill all of his required duties, this process provides documentation of objective measures in the event the administrator is to be disciplined or terminated.

Conversely, this approach can also help good administrators who have poor managers. It allows a formalization that indicates how well the administrator is doing, and requires the manager to consider the requirements for the position and what should be done for planning (via the goals process). This protects the administrator from personality conflicts with their manager by providing a semi-objective measure of their performance. Finally, in the case of a poor administrator with a poor manager, it was felt that in most cases this approach would not improve an already bad situation, and that neither administrator nor manager would be willing to adopt it due to the accountability and effort requirements.

Comments from the Field

When speaking with a manager we knew, his comments were centered primarily on the first stage of the review, the goal setting portion. His concern was that he would rather not set goals, then come back to them some time later and ask if they were met or not. Instead, he would prefer a mechanism that would allow him to ensure that all goals set were on track for being met.

The authors agree with him completely on this and note that, while a performance review often happens yearly, proper management happens daily. In this spirit, the goal section is not intended to be done in one year chunks, but rather should be a process that occurs approximately every three or four months, at an interval that is sensible for the organization. It is especially important in system administration that this process is visited often, as often the major tasks to be performed can not easily be predicted for an entire year. It should also be noted that, while the goal setting and goal review process happens every few months, a good manager will follow up regularly with his employees to ensure that they are on track to meet their goals and to provide any assistance they need.

When speaking with an administrator, comments focused on the pros and cons of evaluation by results. Using the time delay for a server install or customer problem resolution, he argued, was somewhat unreliable. His thoughts were that the most valid method of measurement was the number of times an administrator revisited a problem that had been previously solved.

In this case, we believe that the time measures can be valid, but must be used in context with the other aspects of the review in order to be interpreted meaningfully.

Conclusions

This paper presents an approach to the creation of performance review standards for the area of system administration. This approach is divided into three parts: achievement of goals, service-level requirements, and general competency. It is organized such that administrators can show that they are meeting a standard level of accomplishment, and that managers can know what they should expect from their administrators.

However, this approach provides a framework only. That is, the actual values for the service level requirements, for example, will be organization dependent, and so will require that appropriate values be

determined by each organization before this performance evaluation can be deployed. Unfortunately, there is no "blue book" of standard values available, such as how long it should take to install various pieces of software, etc.

This paper was written to address what the authors perceived as a lack of information for the system administration context. We invite feedback on this approach and encourage discussion and further publications on the subject.

Author Information

Carrie Gates began her system administration career as the sole administrator at a small not-for-profit organization. She left that position for a similar position at Dalhousie University, where she later became the System Manager for the Faculty of Computer Science. She held the manager position for three years before leaving to pursue a Ph.D., specializing in network security. She is currently in the final research phase of her degree, and can be reached at gates@cs.dal.ca.

Jason Rouse has been a system and network administrator for a wide range of companies, such as universities and private businesses, in both Canada and Holland. Most recently he was a Systems and Security Architect for a small private company in Halifax, Canada. He is currently pursuing a Masters degree specializing in networks and network management, and can be reached at rouse@cs.dal.ca.

References

[1] Donohue, Gene, Creating S. M. A. R. T. Goals, http://www.topachievement.com/smart.html, Last visited: 20 May 2003.

[2] King, Patricia, *Performance Planning and Appraisal: A How-To Book for Managers*, McGraw-Hill Book Company, 1984.

[3] Kirkpatrick, Donald L., *How to Improve Performance Through Appraisal and Coaching*, AMACOM Publishing, New York, 1982.

[4] Kubicki, Carol, "The System Administration Maturity Model – SAMM," *Proceedings of the Seventh System Administration Conference (LISA 1993)*, Usenix Association, Monterey, California, November, 1993.

[5] Limoncelli, Thomas A. and Christine Hogan, *The Practice of System and Network Administration*, Addison-Wesley Publishing Company, 2001.

[6] Phillips, Gretchen, *Hiring System Administrators*, Usenix Association, 1999.

[7] Ringel, Matthew F. and Thomas A. Limoncelli, "Adverse Termination Procedures -or- 'how to fire a system administrator'," *Proceedings of the 13th Systems Administration Conference (LISA 1999)*, Seattle, Washington, USA, 1999.

[8] Rouillard, Larrie A., *Goals and Goal Setting: Achieving Measured Objectives*, Third Edition. Crisp Publications, 2003.

[9] Smith, Douglas K., *Make Success Measurable!*, John Wiley & Sons, New York, 1999.

Peer Certification: Techniques and Tools for Reducing System Admin Support Burdens while Improving Customer Service

Stacy Purcell, Sally Hambridge, David Armstrong, Tod Oace, Matt Baker, and Jeff Sedayao – Intel Corp.

ABSTRACT

System administrators are under pressure to do more work and provide better customer service with fewer staff members. At the same time, other challenges emerge: constant interrupts, poor morale, career development needs. At Intel Online Services, we use peer certification to reduce system and network administration burdens while simultaneously improving both customer service and staff morale. Intel Online Services (IOS) has teams of system administrators specializing in various areas such as Virtual Private Networks (VPNs), mail, DNS, and firewalls. Before **peer certification** these specialists did all of an area's work, from completing routine changes and handling problem escalations, to doing engineering work. Peer certification was created as a way to add qualified personnel. Specialists certified their peers by having them pass oral content tests and by supervising them doing changes. Tools were created to simplify administration tasks and make them doable by nonspecialists, and varied in complexity and flexibility depending on the expertise needed to do the task. After implementing peer certification, the number of staff certified to make basic changes increased greatly, along with the number of changes made by front line staff, while the number of escalations decreased. Morale improved as interrupts were reduced and staff gained new areas to learn while customer issues and requests were resolved more quickly.

Introduction

In today's current economic environment, system administrators are under pressure to do more work, often with fewer staff members. Even as budgets are cut and staff are laid off, competitive pressures demand that system administrators deliver better customer service. Along the way, other challenges emerge: constant interrupts, poor morale, career development needs. At Intel Online Services, we used **peer certification** to reduce system and network administration burdens while simultaneously improving both customer service and staff morale. This paper describes the process of peer certification, the tools and automation needed to support it, and our experiences implementing it.

Intel Online Services (IOS) has teams of system administrators specializing in various areas such as Virtual Private Networks (VPNs), mail, DNS, and firewalls. Before peer certification these specialists did all of an area's work, from completing routine changes and handling problem escalations, to doing engineering work. In addition, IOS's configuration tools were powerful yet arcane and cryptic, making them difficult to use by nonspecialists. The first sections of the paper describe our problem environment and the requirements that we had for a certification program and for tools.

We created peer certification as a way to add qualified personnel. In addition to procedures and content tests, we also created tools to simplify administration tasks and make them doable by nonspecialists. The next sections of the paper cover our certification process and tool implementation.

After implementing peer certification, the number of staff certified to make basic changes increased greatly, along with the number of changes made by front line staff, while the number of escalations decreased. Morale improved as interrupts were reduced and staff gained new areas to learn while customer issues and requests were resolved more quickly. We examine our results in the next section of the paper. One thing we learned is that peer certification works better in some kinds of environments and for some kinds of problems than others. The final section of the paper discusses where peer certification is most appropriate and what to look for in tools.

Problem Environment

Why did we need peer certification? Intel Online Services (IOS) is a provider of managed web hosting services. IOS has teams of system administrators specializing in various areas such as Virtual Private Networks (VPNs), mail, DNS, and firewalls. Before peer certification, these specialists did all of an area's work, from completing routine changes, handling problem escalations, to doing engineering work. In addition, our system configuration tools had powerful but

arcane command line oriented interfaces [1] or equally cryptic graphical user interfaces.

The complexity, flexibility, and nonintegrated nature of these tools created a number of problems. They were generally arcane enough so that they were unusable by operations/help desk staff and other system administrators not expert in those specialties. Even for specialists, the interfaces to the utilities were complex enough to lead to mistakes. Some changes and troubleshooting tasks required manipulating multiple files and tools, lengthening the time it took to do those tasks as administrators were forced to change from tool or file to tool in order to complete the work. Switching like this also made it easy to make simple mistakes. As an example, let's look at some of the steps necessary to change a domain in DNS.

1. Locate domain file.
2. Open domain file for editing.
3. Add proper records to domain file.
4. Edit domain serial number to reflect date.
5. Save the domain file.
6. Execute ndc reload command with domain name.

Figure 1: Steps to change a domain in DNS.

Figure 1 shows that a system admin first needs to find the file where the domain is located. If he has appropriate access, he can open the file to add, change, or delete the proper records. Next, he has to edit the domain's serial number in a way that reflects the date. After saving the file, he needs to execute the ndc reload command. Moving from step to step takes time. Several of the steps provide opportunity to make simple errors. For example, text editors typically do not check configuration file syntax. They certainly do not check for non syntax errors such as setting the serial number/date far into the future.

At the same time, customer service suffered as customers had to wait for problems and changes to be handed off to specialists and then wait for results to be communicated back. If a problem or change reached a specialist administrator and more information or clarification was needed, even more time was lost contacting the customer.

This situation hurt morale. System administrators were constantly interrupted by routine changes and simple problem escalations (on call escalation reached up to 50 pages a week). More interesting tasks, such as evaluating new equipment or implementing new systems, were pushed aside by these interruptions. For their part, help desk staff were frustrated that they simply passed on requests and had no way to address them by themselves.

Here is a summary of our key problems:

1. Available tools made troubleshooting and change implementation doable only by limited group of specialists.
2. Available tools made troubleshooting and change implementation time consuming and made making errors easy.
3. Escalation handoffs led to extra time and potential errors.
4. Constant interrupts and escalations affected morale and productivity of admin teams.
5. Lack of ability to make changes proved frustrating for help desk staff and other administration.
6. Customer service, particularly the time to troubleshoot problems and make changes, needed to improve.

Certification Requirements

IOS clearly needed more people who could do routine changes and debug problems while being able to respond quickly to customers. Because of economic conditions, existing staff, usually other system administrators or help desk and operations personnel, was the only source of additional people. We needed a way to get our staff quickly up to a level of expertise that would allow them to quickly and correctly handle the most frequent changes and escalations, and we had to make sure that they were ready before we let them work in a particular area.

To make sure that staff were able to perform necessary tasks, we created a program to certify them as ready. To be effective, our certification programs had to meet a number of requirements. The first requirement was that certified staff could do the set of tasks we needed done and do them correctly. Having more staff but having them make many mistakes would be counterproductive. Also, a person might have a certification from an external entity, but that did not mean that they were able to effectively use the tools in our data centers or that they were effective at all [2].

Next, our certification processes needed to deal with the fact that different tasks have different expertise requirements. Some changes or basic troubleshooting were relatively simple, while others required more knowledge and experience.

Finally, our certification program had to effectively deal with our workload. We knew that there were certain changes and troubleshooting requests that occurred most frequently and took most of the time. That fact implied that the Pareto Principle [3, 4] was at work. Pareto analysis is used as a quality tool through much of Intel. The Pareto principle says that about 80% of all requests (or problems) are from 20% of request types. For example, in the DNS arena, most of the change requests we received were modifications to existing domain or requests to host new domains. Other requests, such secondarying domains or adjusting BIND parameters, did occur, but much less frequently. Thus our certification programs had to cover the 20% of tasks that make up the 80% of requests. The Pareto Principle tells us that if we did not deal that critical 20% of requests, we would not be

reducing our workload significantly. It also implies that dealing with the remaining 80% of request types has diminishing returns. The Pareto Principle became very important in terms of where to invest our development energies and what benefits we would get.

Tool Requirements

It was very clear that the way that we implemented changes and did troubleshooting would have to change. The fact that our certification program would increase the number of people doing changes created new demands. The old way of doing things was to work out of root level accounts. If there were problems caused by a change, we knew that only a few people had that kind of access, and we could contact them.

With the prospect of many more people doing changes, our tools would need to authenticate who was making changes. Once authenticated, the tools should only allow changes that the tool user was certified to make. In case there were problems, we wanted audit trails to track who made what changes, and we wanted a way to back out changes if possible.

To enable staff to get up to speed quickly, our tools had to be easy to use and easy to learn to use. We also had to make sure that that tools prevented errors from being made. Finally, since we did not have a dedicated tools development team, the system administrators in each area would need to develop their own tools. We summarize our tool requirements as follows:

1. User authentication
2. Limiting tasks to those certified
3. Audit trails of changes made
4. Ability to back out changes
5. Easy to use
6. Easy to learn to use
7. Error prevention
8. Implemented by system administrators

Implementing Peer Certification

The first step in implementing peer certification was to partition changes and troubleshooting into distinct specialties. The obvious place to way to group tasks was by system administrator specialty, so we created certifications programs for mail, DNS, VPN, bandwidth management, and firewall access control lists.

The next step was to create three levels of tasks within each specialty: basic, intermediate, and advanced. Basic contains the most common and straightforward change and debugging tasks. Intermediate level means that more complex changes and troubleshooting can be done. Advanced is the highest level of competence, typically involving deeper levels of configuration, such as modifying sendmail [5] macro files or altering BIND [6] parameters.

One of our goals in creating these distinct certification areas and levels was to create achievable milestones. While we could have created a basic certification that covered VPN, mail, and DNS, that would have been harder for staff to achieve because of the breadth required.

In a similar fashion, we could have created a DNS certification that covered basic through advanced knowledge, but that would have been equally difficult to achieve. Partitioning certifications in this manner also provides paths with recognized milestones that staff can follow to improve their skills and their careers.

Another key goal in partitioning expertise was to get most of the frequent and easy changes and problems covered in basic level, keeping in mind the Pareto Principle. Creating expertise levels was also our way of dealing with the fact that different kinds of customer changes and debugging required different level of expertise. Figure 2 shows examples of different levels of tasks in the DNS area, and Figure 3 shows examples in the e-mail area.

1. Basic – Edit customer domains, provide zone information to customers
2. Intermediate – Create new domains, secondary domains
3. Advanced – Modify BIND daemon configuration parameters, adjust zone parameters

Figure 2: Examples of different levels of tasks in the DNS area.

1. Basic – Debug common mail problems using the log search utility
2. Intermediate – Set up aliases and virtual mail hosting for customers
3. Advanced – Edit sendmail .mc files.

Figure 3: Examples in the e-mail area.

To pass a certification level, a staff member usually needed to meet three requirements: passing a test on the specialty at the appropriate level, doing a certain number of supervised changes, and being certified at lower levels of certification if any (i.e., you must pass basic and intermediate before doing advanced). Test questions at each level were created to make sure a person being certified was competent at the specialty's area. These questions and all training materials were available online to provide access at any time and to anyone on any shift.

An area's system administrators conduct the tests orally, using the provided questions as a starting point. The testers were free to change the questions or ask more details based on those questions in order to make sure that the test taker really understands the subject and doesn't merely "parrot" back answers. This is one key advantage of the peer certification method over more formalized and rigid certification systems. Rather than relying on just a set number of questions, system administrator specialists can add or alter the questions to make sure that their peers comprehend

what is important and relevant. They have incentive to pass people in order to reduce workload, but since they handle escalations, they also have the incentive to make sure that people know their material and can do work correctly.

After the subject matter test, a total of four changes or problem troubleshooting need to be performed. This way, we ensure that the person being certified can actually do useful work. This is analogous to driving or flight experience. To get a drivers license, in addition to passing a test, you need to pass a road test. The changes are supervised by one of an area's specialists. Once again, this allows us to make sure that the person being certified is competent.

When a person passed an area certification, we notified the person's manager and announced that fact to the system administration team for that area and often to the entire data center, and added their name to a web page containing the list of people certified to make changes. This was intended to inform relevant staff that a new person was available to do work in an area. It also served as recognition and a reward for that person's work.

Tool Implementation

The next step was to create tools that would allow people with various levels of skill to safely do changes. We didn't have a dedicated group of tool developers, so all programming had to be done by the area specialists. To make the tools as accessible as possible, most of the tools we built were used via the web as CGI programs. Using of scripting languages such Perl [7] and Expect [8] helped us rapidly develop tools. Sharing code also helped speed up development. Since many of the tools involved using a web interface to modify text configuration files, some tools were converted into others. For example, the DNS zone tool editor originated as the ACL web tool.

Tool flexibility varied with the skill level (see Figure 4). The more expert the level, the more flexibility and choices the tools allowed; the less expert the level, the less flexibility allowed. This is to prevent staff with only basic expertise from making mistakes while granting more flexibility to the more expert.

We had requirements to log changes, provide an audit trail, and backout mechanisms. Users were authenticated using TACACS [9] and Radius [10]. To track changes, the changes, represented as file diffs, are sent to a mailing list of system administrators. The messages contain the ID of the person making changes. In addition, the mailing list is archived using hypermail [11], which makes it easy to browse through changes that have been made. Configuration files are checked into RCS [12] to make sure that we can recover older versions, and the ID of the change implementer is logged in the comment field.

Security proved to be a major concern, as mistakes in or sabotage of a customer's domain or their firewall configuration could be devastating. We had to safely execute certain functions (such as BIND's ndc program) as root, despite CGI's running under Apache run as user www. To do this, we resorted to setuid C programs that call other programs. Other security precautions include keeping direct input from the user to a minimum and carefully parsing input that we do receive to ensure that no dangerous input is accepted, such as shell escape characters.

The infrastructure needed for our tools was fairly simple, as shown in Figure 5. In each data center, we ran our web tools on a change server that was permitted to access and to make changes to network equipment and servers. This enabled our tools to share configuration information and take advantage of authentication infrastructure. There were separate authentication servers for TACACS and RADIUS. We also had a reporting server that continually took extracts of data and summarized

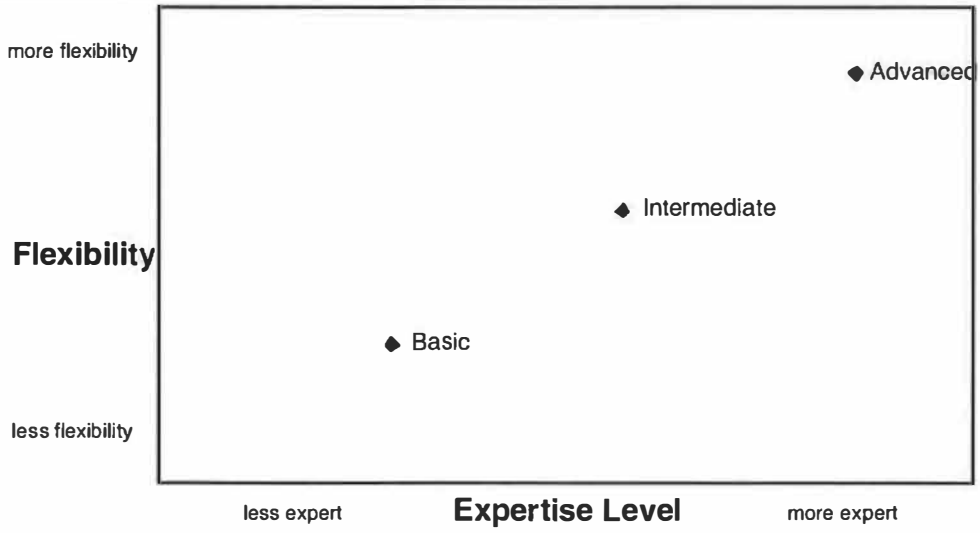


Figure 4: Flexibility vs. skill level.

them. Some tools ran on individual managed servers and were accessed directly through the web.

Results

We created tools to simplify change implementation and troubleshooting, allowing the number of staff certified to make basic DNS, VPN, and firewall changes to increase greatly (tripling in some cases). These tools and our certification program increased the number of problems resolved on the first call (without escalation or handoff). Figure 6 shows the number of escalations taken by the system administrator groups working on Firewall ACLs, DNS, mail, and VPN.

We started peer certification around work week 22 of 2002. Escalations peaked at 50 escalation during work week 26, and the number of pages started falling off as staff became certified and began doing changes and troubleshooting. Customer service simultaneously improved as changes and problems were turned around faster since the need to escalation was eliminated for most changes.

In one particular IOS data center, the percentage of DNS changes made by help desk/operations staff increased from 0% to 70%, close to that predicted by the Pareto Principle. As was experienced in a number of other environments, the certification programs increased morale. Help desk staff could now resolve problems directly and had opportunities to learn new skills and knowledge. They appreciated being recognized by their management and peers when they passed certification levels; this proved to be a cheap but effective reward system. Not only did system administrators enjoy the reduction in routine, simple work, but they also had new opportunities to learn skills in areas new to them.

Our tools generally worked well. We developed tools that did allowed staff with varying skills levels to

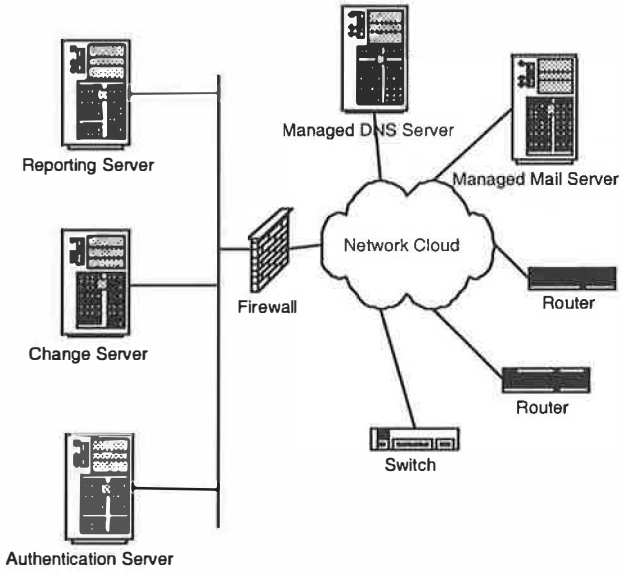


Figure 5: Tool infrastructure.

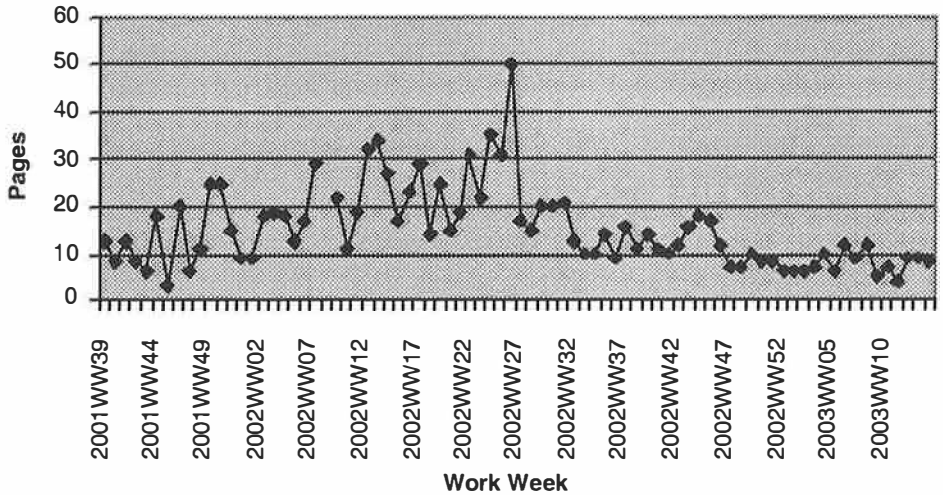


Figure 6: Escalation pages per work week.

do things like change VPN passwords, change customer DNS domains, and do firewall ACL changes. System administrators created all of these tools, and this proved advantageous for several reasons. First, we didn't have a dedicated tool development team or funding to outsource their creation, so system administrators were the only people who could create them.

Second, this sped development, as communication loop between developer and requester, along with the opportunities for misunderstandings and misinterpretations of requirements, was eliminated. The people who understand the problem the best were creating the tools. Also, using scripting languages like Perl allowed more than just a small select group of people to look at the tool source code and identify and fix problems. This would have been more difficult if all of the tools were built in C, C++, or Java. Finally, as mentioned above, the system administrators had a strong incentive toward making this work correctly, since they would be escalated to if there were problems.

Our certification processes and tools worked well for the most part, but there were a number of issues and problems that did come up. One surprising occurrence was that more people did not take the opportunity to become certified. Figure 7 shows who did basic DNS changes in our largest data center during from December 2002 through April 2003.

Staff Type	Number of Changes	Percentage
Core DNS staff	14	18.2%
Help Desk staff	43	55.8%
Other administration specialists	20	26.0%

Figure 7: Breakdown of DNS change ownership.

Figure 7 shows that 55.8% of changes were done by first line staff. We thought that help desk staff would try for every certification available, but we found out that motivation varied from person to person and from shift to shift. While first call resolution improved overall, there were certain shifts that did not have coverage for the basic certification in all areas. As a result, management in some data centers required their staff to become certified.

In other cases, some data centers seemed quite content to simply pass on requests. To some of our staff, the testing phase proved extremely intimidating. We had staff with test phobias, and while we knew that they could probably pass some of the tests, they elected not to take them. As a result, they were not certified. On the other hand, specialists in other systems administrator areas tended to take advantage of the certification programs. In general, they seemed to have a much higher motivation to learn new things and grow – a pattern that we saw in other data centers.

Figure 7 has good news for the core DNS staff where they once did 100% of DNS changes, they only had to do 18.2%. Ideally they would have to do 0%, but we encountered the obstacles mentioned above.

One problem that we encountered was getting enough changes and troubleshooting opportunities for people to get certified. Sometimes this was because of our own success. Certified front-line staff would do all of the changes and not leave any changes for people being certified. Shifts and time zones caused problems also. Some shifts often had few or no available area specialists to supervise changes, particularly night shifts at locations with most of the system administrators. In these cases, special arrangements had to be made for certification purposes. In some cases, the lack of changes was simply because of the breadth and depth of areas that we tried to create certifications.

Going by the Pareto principle, 80% of the changes and problems would be taken up by some 20% of the total types of changes and problems. Once we had covered those 20%, getting the remaining 80% would be distributed among 20% of the incoming requests. In our zeal to create certifications, we didn't always keep the principle in mind, and thus our time was poorly spent creating certifications in areas that had relatively few requests.

For some tasks that did not require a lot of upfront knowledge, our certification process was not used. The VPN group created a very simple interface for updating passwords. There was not a lot of knowledge needed to use it, so the tool was deployed without requiring certification. The VPN administrators gave a brief training session to prospective users, issued user IDs, and gave permission to start using the tool. Similarly, a tool was created by our network group to move systems from one Virtual LAN (VLAN) segment to another. The interface was very simple, and all the change implementer needed to know was the current and destination VLAN number. Access was also given after a brief training session.

For the most part, our tools worked well, but there were a number of areas that proved to be problematic. One area was the problem of "leaky abstractions" [13]. Our tools (and indeed, many automation tools) abstract and hide all of the mechanics of a change. For example, our DNS change tool accepts changes to a zone, edits a zone file, and then forces a reload of that zone file. To the help desk staffer making the change, all the change he or she does is bring up the domain, fill out a web form, and click on a "propagate" button. DNS zone changes have been abstracted to simply filling in a form, and the user assumes that the reload was effective. If there were a problem with loading the zone, that information would never make it back to the change

implementer. This is one example where the abstraction broke, and problems leaked through.

We had problems where our tools were too flexible, despite the precautions we took. By "too flexible" we mean that the tools allowed enough choices and ambiguity to cause problems. Our DNS change tool allowed the addition of DNS records without a record name. The default record name for an entry in BIND zone files is the previous record, and the zone if no records have been specified. Because of the way that many of our zone files looked, some change implementers assumed that the default record name of such entries was the domain name. This assumption caused a number of problems, which could have been avoided if the tool was less flexible and did not permit the addition of entries without record names.

Another tool problem we experienced involved the impact of our tools. We developed tools for our mail relays to measure and graphs the load average and mail queue length, and the top destinations for which mail is queued. We also developed a tool for browsing mail server logs. These tools were available via the Web in order to make them accessible to help desk staff and other system administrators. The problems occurred when the mail relays became heavily loaded. That was the time when we were very interested in the load average, mail queue length, and what were the top destinations in the mail queue.

Analyzing mail queue contents and checking mail logs caused significant additional load and made loading problems even worse. The lesson we learned here was that debugging tools should not make problems worse. A better implementation of the tools would have moved data off the servers where they could have been analyzed without impact.

Finally, we had the opportunity to push changes even closer to the customer by creating tools for them to make their own changes. Unlike staff, we couldn't certify our customers, so these tools had to be as simple and foolproof as possible. With end users, the flexibility vs expertise chart looks like Figure 8.

Our best success was a narrowly focused tool that allowed the customer to make specific DNS changes: changing an MX record and adding a certain CNAME record. The tool only allowed the customer to select a zone and then push buttons that did operations. This tool saved IOS staff from doing several changes a week. We had less success with a tool for converting DNS zones to use a distributed content vendor. The tool had multiple functions, such as browsing zones, and converting zones. It accepted domains in some forms and entire fully qualified domains names in other forms, and this caused confusion and questions from the customers. Again, excessive flexibility and ambiguity in interfaces caused problems.

During mid 2003, Intel left the web hosting business, moving all of the customers and the current environment to another provider. Peer certification made transition control of the environment much easier. In order to create the peer certification process in a system administration area, we had to document our environment and tools, create training for use of the tools, and have the documentation and training on-line and ready for anyone who wanted to learn.

Applying Peer Certification Techniques in Your Environment

The applicability of peer certification to other system administration environments depends on the conditions in those environments. The process of doing peer certification can be a weighty one, requiring testing and monitoring of a certain number of changes. From our experience, tasks that require some knowledge before changes are best served with a

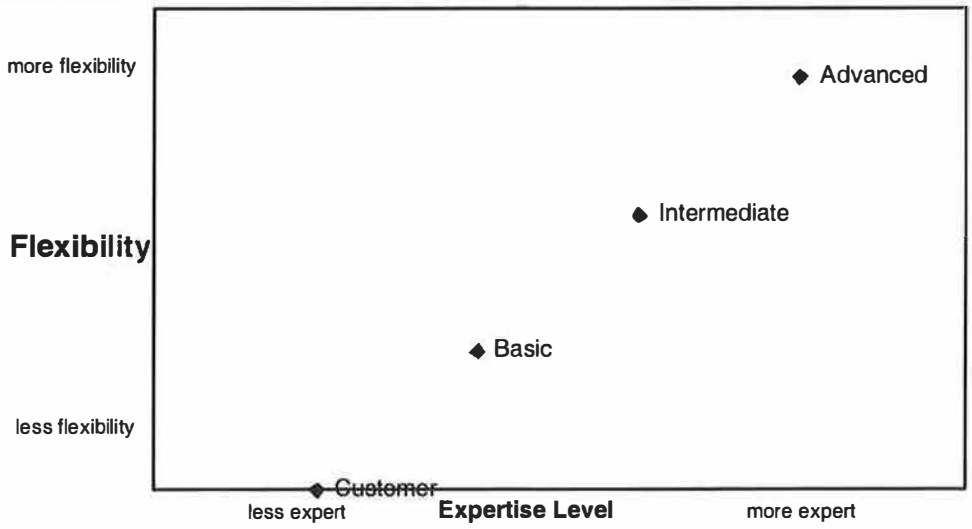


Figure 8: Flexibility vs. expertise level.

certification process. For example, no matter how easy a DNS zone modification web interface may be, the change implementer still needs to know what DNS is and the characteristics for different records. Mail debugging requires specialized knowledge, as do firewall rule changes. On the other hand, a task like doing password changes is fairly simple, and peer certification would be overkill.

Staffing environments are another consideration in applying peer certification. Environments where there are a lot of handoffs of problems or changes – either moving tasks to someone's work queue or escalating problems – are ideal for peer certification. These situations often involve help desks or call centers and present an opportunity to reduce handoffs and escalations, improve the time to service calls, and improve working conditions. On the other hand, a very small shop where a small staff handles almost everything would not benefit from peer certification, since most everyone would know everything anyway and the certification efforts would not gain much.

A very large environment would most likely benefit from peer certification, although certification work would have to go on continuously with staff turnover. Peer certification in a large environment would encounter some of the challenges we experienced, particularly dealing with many different zones and shifts. System administrators would have to be prepared to occasionally work at odd hours to supervise certifications, but at the same time, peer certification could be used to train other system administrators to do this part of the process.

To deploy peer certification effectively, web accessible tools that are easy to learn and use are essential. While tools we developed at Intel are not publicly available, there are a number of extensible tools that do similar functions, such as the Los Task Request System [14], Webmin [15], and Linuxconf [16]. We were not aware of these tools (a common problem [17]) before writing our own. Before using these kind of web based system administration, you will need to make sure that they have proper authentication, access controls, and audit trails. In addition, extensibility is very important, as you will likely need to write tools or modules to fit your own environment.

No matter what size the staff, some tool development and analysis techniques we used are usable in any environment. Pareto techniques are a great way to maximize the effectiveness of any automation effort. Charting what problems or changes occur most frequently and take the most time reveals where efforts will be most effective. Automating multiple step changes or functions into a single step is a time saver and can even persuade experienced system administrators to use automation tools.

Having system administrators choose or develop their own tools using scripting languages is an excellent strategy. It eliminates communication delays and misunderstandings that might occur between system administrators and a separate developer. Making tools and documentation available from a web browser opens up access, and should be coupled with making tool flexibility vary depending on the ability of the tool users. Any tool that implements changes needs to authenticate users and allow them to change what they are permitted, and there needs to be backout mechanisms. Finally, any monitoring tool should avoid negatively affecting what it is monitoring.

Conclusions

By increasing the number of qualified personnel, peer certification reduced system administrator burdens at Intel Online Services while improving customer support and increasing staff morale. Implementing peer certification requires defining tasks appropriate to differing levels of expertise, creating tests that measure competence, and creating automation tools that simplify and safeguard the process of making changes or debugging problems. Peer certification can help the most in environments where tasks have many handoffs before being done and those tasks require detailed knowledge in order to accomplish them. Areas to carefully consider when using peer certification are dealing with differing motivation levels, preventing leaky abstractions in tools, and making tools with appropriate flexibility.

Author Information

Sally Hambridge has worked for Intel Corporation since 1984, where she was a librarian, a database administrator, and a system administrator. She was with Intel Online Services from 1999 to 2003 as a Firewall Engineer and was team lead for the ACL team. She is reachable at sallyh@ludwig.sc.intel.com.

Stacy Purcell graduated from the Georgia Institute of Technology with a BS-CS in 1992. He joined Intel Corporation in Folsom, CA immediately after graduating where he has been employed in several roles including System Administrator, Network Engineer, and manager over the course of the last 11 years. Reach him electronically at stacy.p.purcell@intel.com .

Jeff Sedayao is a network engineer for Intel. Between 1987 and 1999, he architected and ran Intel's Internet connectivity, and after that, worked in Intel Online Services. His primary interests are in Internet performance, security, and policy implementation. Jeff can be reached electronically at jeff.sedayao@intel.com .

Tod Oace is a UNIX and networking systems engineer for Intel. Over the past 20 years he has worked with a wide range of computer platforms from micro to mainframe, and is proficient in several programming languages and networking protocols. Tod worked in the Firewall Engineering group of Intel Online Services. His email address is tod.r.oace@intel.com .

David Armstrong is a Network and Firewall Engineer. From 2000 to 2003, he worked in at Intel Online Services in the Firewall group, specializing in Firewalls, VPN connectivity, and DNS.

Matt Baker is a technical marketing engineer with Intel's Platform Networking Group. From 1998 through early 2000, Matt led Intel's broadband and VPN technology trials, designing and deploying one of the earliest large scale corporate xDSL and VPN remote-access networks. At Intel Online Services, Matt focused on the concept of the Datacenter, a place where internet service provider and enterprise issues can frequently converge and how these issues affect VPN connectivity/performance, network security design, and AAA systems design. He can be reached at Matt.W.Baker@intel.com .

References

[1] Hambridge, Sally L., Tod Oace, Jeff Sedayao, and Charles Smothers, "Just Type Make! Managing Firewall Using Make and Other Publicly Available Utilities," *First Usenix Conference on Network Administration*, Santa Clara, CA, April 1999.

[2] Vaas, Lisa, "IT Paper Chase," E-Week, http://www.eweek.com/article/0,3658,s%253D703%2526a%253D13923,00.asp , September 3, 2001.

[3] Donnini, Alex and Alan Miller, "Relieving the Burden of System Administration through Support Automation," *LISA 14 Proceedings*," December, 2000.

[4] Slater, D., "Call Center Management," *CIO Magazine*, http://www.cio.com/archive/040100_numbers.html , April, 2000.

[5] http://www.sendmail.org/ .

[6] http://www.isc.org/products/BIND/ .

[7] http://www.perl.org/ .

[8] Libes, Don, *Exploring Expect,* O'Reilly and Associates, Inc., Sebastopol, CA, 1995.

[9] Cisco Corporation, *TACACS+*, http://www.cisco.com/en/US/customer/tech/tk583/tk642/tech_protocol_family_home.html .

[10] Rigney, C., S. Willens, A. Rubens, and W. Simpson, "Remote Authentication Dial In User Service (RADIUS)," *RFC 2865*, June, 2000.

[11] http://www.hypermail.org/ .

[12] Tichy, W. F., "RCS – A System for Version Control," *Software Practice & Experience*, Vol. 15, Num. 7, July, 1985, pp. 637-654.

[13] Spolsky, Joel, "The Law of Leaky Abstractions," *Joel on Software*, http://www. joelonsoftware.com/articles/LeakyAbstractions.html , November 11, 2002.

[14] Stepleton, Thomas, "Work-Augmented Laziness with the Los Task Request System," *Proceedings of LISA 2002: Sixteenth Usenix System Administration Conference*, November, 2002.

[15] Cameron, Jamie, et al., "Webmin," http://www.webmin.com/.

[16] Solucop, et al., "Linuxconf," http://www.solucorp.qc.ca/linuxconf/ .

[17] Burgess, Mark, "System Administration Research," *;login*, June, 2000.

ISconf: Theory, Practice, and Beyond

Luke Kanies – Reductive Consulting, LLC

ABSTRACT

ISconf is a configuration management system (CMS) developed through years of experience automating configuration management. As the number of CMS's available for use increases, attempts have been made to explain why one should use ISconf instead of those other tools; specifically, a stark contrast has been made between tools which depend on *convergence*, like cfengine, and ISconf, which strives for *congruence*. This paper discusses production experience with ISconf, what was good and what was bad, and also analyses recently developed theories used to explain why ISconf might be superior to other tools. It further discusses experience integrating ISconf with cfengine, a process in direct conflict with the accepted ground rules for using ISconf. Based upon this experience, there are serious limitations in practical usage of ISconf, and proposed theories that support its use are insufficient and unconvincing. Although ISconf has been found to be a useful tool, it cannot be considered a sufficiently useful or powerful answer to arbitrary configuration management needs on its own.

Introduction

ISconf [isconf] began life as a makefile [make], as described in *Boostrapping an Infrastructure* by Traugott, et al., [bootstrap]. It was a file specifically ordered and arranged to describe the dependencies and relationships between different hosts and the work that needed to be done on those hosts. Every host was listed as a make target and had a list of dependencies associated with it; one could simply run make -f isconf.mk <hostname>, and make would verify that all of those dependencies had been resolved. This is a very simple concept; Unix system administration can usually be broken down entirely into a sequence of command lines, and this structure merely took advantage of that by maintaining that list of command lines in the order they were executed on a given host.

Although ISconf has gone through two full rewrites since that initial version, it has barely changed at all in practice. It still functions entirely as a means of associating a list of command lines with a given host name, and that list of command lines is still contained in a makefile; all that has changed is how the order of commands is maintained and how the mapping between host names and commands is maintained.

What has changed the most about ISconf in that intervening time is the theory behind it. While it was originally developed out of practice, and its usefulness hinged entirely on experience, attempts have been made to provide a theoretical basis for ISconf's superiority as a tool [order]. While these attempts at theory cannot lessen ISconf's basic usefulness, they have served to muddle the practice of using ISconf, because of that theory's requirement that ISconf not be used in conjunction with any other CMS.

Recent Developments in Theory, and Their Consequences

Initial usage of ISconf demonstrated that it was very simple to develop a list of commands used to build a system, and that if those commands were replayed in the exact order on another host, a host with the same traits as the first could be easily and consistently built. This was found to be very useful, both for disaster recovery and for duplication of system configurations. The theoretical developments around ISconf call this *deterministic ordering*, because they state that the ordered list must be executed in that specific order every time, with no deviation.

This deterministic ordering requires three features that all known versions of make possess: State maintenance, failure on error, and consistent ordering. This has allowed make to be the engine for all of ISconf's functionality with minimal development. Unfortunately, this ordering requirement also introduces a requirement that make cannot meet: any failures must leave the system completely unmodified and in a recoverable, consistent state, meaning that successful steps attempted as part of an unsuccessful sequence must be backed out completely, leaving the system as though the sequence had never been begun. This concept is called *atomicity* [prolog].

State Maintenance

The functionality of make centers around knowing how to build programs which rely on various input files. One develops a list of commands, known as a *stanza*, named for the file one wants to create; if that file does not exist, or is out of date, then those commands should be run. If the commands do not create the named file, then the commands will be run again when make is executed again.

ISconf has a special *stamps* directory devoted to files marking completed commands. All make stanzas executed by ISconf perform their appointed task and then put a stamp file in this directory to mark that they have completed successfully. Given a list of dependencies associated with a host, make will only execute those dependences for which a stamp file does not

exist; therefore, make is maintaining a representation of the state that the host is in. Here is an example stanza, to elucidate this process:

```
enable_ftp:
  cp /etc/inetd.conf /tmp/inetd.conf
  sed 's/#ftp/ftp/' /tmp/inetd.conf \
                    > /etc/inetd.conf
  /etc/init.d/inetd restart
  touch $@
```

This stanza makes a backup of the config file to be modified, then uses sed to uncomment ftp from that backup file and overwrite the main configuration file. Inetd is then restarted. Finally, a file with the same name as the stanza (the macro $@ refers to the name of the stanza being executed in make) is created so the stanza does not get run again.

Failure on Error

As make works its way through a list of commands to execute, it exits immediately if any of those commands fail. This fits its original purpose well, because it is assumed that some later file requires the file that just failed to be built. This assumption works well with ISconf, also; if a command fails, then we want ISconf to exit immediately, because it is likely that some later command will depend on the command that just failed.

Thus the fact that make exits immediately upon encountering an error is used in ISconf to maintain ordering; if a command fails but the next command is executed, we will be executing our command list in an order different from that specified in the configuration. Having this failure on error guarantees that all of our hosts have executed up to the point where they completed their command list or they encountered an error, but in no instance did they skip a step or execute a step out of order.

Consistent Ordering

Configuration files for make, also called *makefiles*, are organized in the form of stanzas, each of which have up to three parts. All stanzas have a name, followed by a colon (:). If the stanza has any prerequisites, they will be on the same line as the stanza name, after the colon. If there are any commands associated with the stanza name, they begin on the next line after the stanza name and are indented. Long lines can be continued with an escaped carriage return.

The make program uses a stable topological sort to determine the execution order of commands specified in its configuration files. Make builds a topology of all parent-child relationships, ordered by the declared dependencies and the order in which they appear in the config files, and then linearly walks through these relationships, executing children first and then parents. As long as the parent-child relationships and the order in which they are specified stay the same, make will execute the commands in the same order every time.

ISconf exploits this consistent ordering. It would be straightforward to develop a tool which consistently performs a list of commands in the exact order they are specified, but make conveniently already has that feature, so ISconf takes advantage of it.

Atomicity

Unfortunately, something that ISconf also requires but that make cannot deliver is the atomicity of all procedures. A set of commands must either complete entirely, or the set must fail entirely and not modify the system at all. If a sequence of commands is begun but a failure is encountered, then the entire sequence should be backed out, so that the system is left in an unmodified state. If this cannot be done, then the system is likely to be in an inconsistent or nonfunctional state. It is possible to make ISconf stanzas idempotent, so that they could be run multiple times with no ill effect, but it is far more difficult and eliminates the simplicity that ISconf provides. If all stanzas were idempotent, then most configuration errors could be easily corrected on the next run of ISconf.

For instance, consider modifying the configuration file of an important service like bind: given a sequence of commands that modify a configuration file and then restart the service, if the restart fails because of a typo in the modification but the modification is not backed out, then the service will be down until a human intervenes.

ISconf requires another level of atomicity; each stanza that ISconf executes must also be atomic. If a stanza has multiple commands that modify a system and that stanza fails, then human intervention will certainly be required to either manually complete the stanza or manually back out the completed portions of the stanza. Given a stanza that creates a directory and then adds it to a system's NFS exports file, consider what happens if the directory creation succeeds but the configuration change fails. Normally one would solve the configuration problem and let ISconf run the stanza again, but the second time the stanza runs, the directory creation fails because the directory already exists. A human must modify the system manually in some way to solve this problem.

Atomicity at both the stanza and the sequence level are required by ISconf, but we see that both are missing, partially because make lacks them and partially because atomicity is a difficult problem [maturity]. ISconf certainly is not any more immune to these problems than other CMS's, and its use of make makes it more susceptible in some ways, especially since atomicity problems can creep into make stanzas in ways which don't show themselves until the stanza has already been run many times.

Practical ISconf Usage

One of ISconf's biggest strengths is that work performed with it is very similar to work performed by hand, so it is very easy to make progress quickly. Unfortunately, whereas a human would naturally take

into account the differences between different hosts, those differences have to be coded into ISconf, and it is impossible to code for all of them. This is where ISconf begins to run into real trouble.

Day to day use of ISconf is affected most by the fact that it relies on make as its execution engine. The best it can hope to do is optimize the mapping between hosts and their execution lists, and possibly simplify some aspects of using make. Most people looking for configuration management systems are seeking some way to better understand the maintenance of their servers, and ISconf is limited in its ability to provide this understanding. ISconf remains useful as long as it is treated as a modest tool with modest goals, but as soon as one begins expecting functionality on par with a full CMS, severe limitations surface.

ISconf's Feature Set

ISconf is an interface to make, and not much else. It is entirely a tool for mapping hosts to commands. If there are functions one needs to perform, those functions must be written. There is no higher-level functionality, no advanced toolset, no powerful configuration language, and no reusable component system. In fact, the ordering that ISconf requires is really the only thing it offers, whereas other configuration management systems usually lack this ordering but provide some semblance of all of the above features. In choosing between ISconf and most other CMS's, this is largely a choice between a tool which easily organizes work and a tool which makes the work itself easier.

It is true that many of these features could be developed over time, especially the advanced toolset. ISconf 3 already has a number of useful utilities. ISconf's use of make as its API to the utilities upon which it depends requires that these utilities must also be capable of functioning independently, because very little information can be passed from ISconf to the utility being executed. This lack of integration also places a lower burden of consistency on the authors of the utilities, which means that utilities are likely to have more localization and individuality than is desirable for a component of a CMS, placing a further burden on the sharing of code. Alva Couch's paper on script maturity [maturity] discusses this topic in more detail.

Because ISconf stanzas depend heavily on the environment for which they were written, the shareable utilities would have to be more abstract than the actual stanzas. So, in order to make it possible to reuse someone else's ISconf code, that code must be written such that it operates irrespective of how it is called by ISconf, which is equivalent to saying it must be able to operate completely independently. Thus, these utilities could be written for any CMS, and cannot be specifically written for ISconf, so ISconf confers no advantage here at all, and is detrimental in that it presents that additional burden of abstraction to those who would like to write shareable utilities.

What ISconf Understands

Similar to how make depends on the writer of the makefile to understand the files it is managing, ISconf depends on its maintainer to understand the commands it is executing. ISconf currently enforces ordering entirely at the command level, not at a symbolic or functional level. This is in stark contrast to tools like psgconf [psgconf], which are specifically developed to manage systems at a symbolic level – much easier for humans to understand – and rely on the CMS to map the symbols into a deployed configuration.

In most ways, this is a limitation of ISconf. At first glance, ISconf makes managing a system as easy as managing a development project, but with use, it quickly becomes more complicated to control. psgconf must understand all files it maintains well enough to rebuild that file from scratch without changing anything else; as a result, an invocation can help one quickly recover from a configuration problem or a lost file.

ISconf provides none of this understanding, which means that more research is required before any change is queued, and any misconfiguration requires time and attention from a system administrator. Consider managing the /etc/services file, which is used for mapping service names to numbers. In psgconf, a specific mapping is maintained in a symbolic form and then converted to a file, so managing this file involves changing this symbolic form, which is very easy for a sysadmin:

```
### Entries for /etc/services in psgconf
port_names {
    "1/tcp" => tcpmux,
    "7/tcp" => echo,
    "7/udp" => echo,
    "9/tcp" => discard,
    "9/udp" => discard,
    "11/tcp" => systat,
      ...
};
```

Any errors in this configuration can be easily repaired, and the file can be instantly regenerated with correct, valid contents.

In ISconf, however, the sysadmin merely queues up a command which does what she thinks is appropriate. If the sysadmin made a typo that was not discovered until the stanza was already deployed, then the file may have to be recovered from backup (during which time the system is in a failure state), or a throwaway stanza must be built to recover from the error; see Listing 1.

You can see that if a few such throwaway stanzas are made, it quickly becomes very difficult to understand exactly how a given ISconf configuration produced the configuration on disk. Most often, the local configuration must be studied, and the list of stanzas to execute (ISconf 3 comes with the utility islist, which provides the ordered execution list for any host) must be followed completely to figure out exactly how a configuration came to be, and thus understand where

to correct a problem. Correcting a problem thus includes reverse-engineering a series of scripts as well as understanding the mapping between configuration and behavior. Because all of a stanza's preconditions are only implicitly documented by the stanzas run previously, a stanza's dependencies cannot be assumed and must instead be tested every time.

This problem becomes even more complicated by ISconf's susceptibility to problems with latent preconditions, where hosts are differentiated in ways that aren't fully understood. For example:

```
add_goodpkg:
  pkginstall GoodPkg
        # adds 'goodpkg 9000/tcp'
        # to /etc/services
  touch $@

add_newservice:
  echo "newservice 9000/tcp" \
              >> /etc/services
  touch $@
```

With the comment attached to the add_goodpkg stanza, it is obvious that these stanzas conflict. But few of the changes done during package installation are obvious, even when the known changes are documented somewhere. If the first stanza is run on a small subset of hosts and the second stanza is run on all hosts, it could take a significant amount of time to track down the source of any problems that result from this conflict. It is possible to overcome this specific problem by using ISconf to generate the /etc/services file similarly to psgconf [psgconf], but this type of problem crops up constantly in more subtle and less manageable forms.

It should be noted that ISconf probably does provide the lowest barrier of entry in terms of managing the /etc/services file, in that it is incredibly easy to modify that file in ISconf, but it provides no extra functionality with that ease of use nor any protections from mistakes, whereas both cfengine [cfengine] and psgconf require more initial investment time upfront but give back significant extra capabilities in return for that investment.

How ISconf Does Its Job

As has already been seen, ISconf relies on make for most of its functionality. Inasmuch as make can be said to have an API (Application Programming Interface), ISconf shares that API. Because make was developed with a very specific purpose, one that is relatively simple when compared to managing operating systems, it has a somewhat limited interface. ISconf inherits all of the limitations of this interface, but tacks some of its own liabilities onto it.

The make program organizes its commands into stanzas; each stanza is essentially self-contained and should perform a single, specific function. ISconf in turn associates these stanzas with hosts and host types. This actually means that ISconf's API is less functional than that of make, because one does not have access to make's full preconfiguration capabilities. Because it is the stanzas that ISconf associates with a host, and not the actual commands, these stanzas function as a mapping between a host and what is done on that host; the commands that a stanza executes could be changed to be something completely different and ISconf would never know, or a stanza name could be changed without changing its functionality and again ISconf would not notice.

What's worse, however, is that this means that each host has two lists associated with it – the list of stanzas and the list of commands actually executed – and neither can ever be modified after the fact. One cannot change the name of a stanza, because make will incorrectly conclude that it is a new stanza and will execute it again. One cannot change the commands that a stanza executes because it is only the old version of the commands that has been tested, and one has no way of really knowing if the new version of the commands will succeed if the host needs to be duplicated or rebuilt. This is a direct result of ISconf managing a list of make stanzas, rather than understanding or controlling what it is actually doing.

This is what really hurts ISconf in the long run. All code and all stanzas that have ever been executed on any host must be maintained until that host no

```
### Entries in /etc/services for ISconf
# if this stanza gets run on all your systems, you are very
# unhappy
add_myservice_bad:
    echo "myservice 888/tcp" > /etc/services  # oops, just overwrote the file
    touch $@

add_myservice_service:
    echo "myservice 888/tcp" >> /etc/services # oops, wrong port number
    touch $@

fix_myservice_service:
    cp /etc/services /tmp/services.bak
    sed 's/myservice 888\/tcp/myservice 8888\/tcp/' \
        /tmp/services.bak > /etc/services
    touch $@
```

Listing 1: Error recovery stanza.

longer exists, even if they are no longer in active use. And even if one has found a new way of managing a given function or file, all of the old methods must be maintained in case they are needed again during disaster or server duplication. This is because all hosts on the network are different in some ways (usually at least hostname and IP address), and although testing can reduce the likelihood of those differences causing problems, it cannot actually eliminate the possibility for problems. Not all dependencies are specifically encoded as such in ISconf; package installation may require a certain amount of free space in /var without there being a check that this space exists. In fact, if this amount of space is proportional to the size of the package being installed, it may not be a problem until an especially large package install is attempted.

Two simple but common examples are presented to illustrate this point, both of them very common functions encountered in usage of ISconf. The first is usage of ISconf to manage packages on Sun Solaris systems. Initial package management was done by the author by creating a stanza for every package that needed to be installed, and then doing all the necessary work in each package.

```
add_cvs:
  mount -t nfs \
      server:/export/pkg /mnt
  pkgadd -A /mnt/pkgadd_noask \
              -d /mnt/cvs all
  touch $@
```

After a few of these stanzas were created, it became clear that some method of reducing code duplication needed to be developed. In response, a single script responsible for all package management was created, as well as a way of referring to that script using wildcards in make.

```
pkg/%:
  mkdir -p pkg
  pkginstall --isconf $@
  stamp $@
```

This script expects a complicated stanza name like pkg/add__java__1.4.1 to be passed to it, and it parses that name to figure out what it is supposed to do. However, because the initial package stanzas were already executed on all Solaris systems, they could never be replaced with this new method. Doing so would risk the new method failing because of its own existing preconditions, such as the new method requiring a package installed using the old method.

This is actually a relatively innocuous example, because the two methods are at least compatible. The author has also encountered situations where an incompatible method of performing a function needed to be developed, such as for managing the contents of a file.

The second example is even more common: Deployment of a script with a subtle logic bug which does not get discovered until it is already deployed. The script works fine on the first hosts on which it is deployed, but when the script is deployed again on a differing set of hosts, it fails in an unforeseen way. The author has written a script for automating usage of Sun's DiskSuite application to mirror a system's boot disk; this script was originally developed entirely on systems with SCSI disks, and thus was only tested on them. When Sun came out with servers with IDE disks, this script failed horribly. However, the theoretical foundations of ISconf required that the script be copied and a new version modified to have the new functionality, rather than merely upgrading the existing version, because again, there is no real way of knowing that the new version would succeed on systems that the old version had already been run on.

This is an example of *software rot*, in that the assumptions of a program became out of date. When the script was originally written, it was not possible to purchase a Sun server with IDE disks, so it was assumed that all disks would be SCSI disks. When Sun began shipping systems with IDE disks, this assumption became invalid. This is a very general problem, because it is not really possible for a script or program to truly understand or state all of its preconditions, and many of those preconditions are only discovered when they are not met in some new deployment, long after the script was first put to use.

These two examples illustrate that ISconf encourages a number of very bad habits: keeping buggy old scripts around, not implementing better management methods merely because they would necessitate what amounts to extra bookkeeping, accumulation of cruft in configuration files, and many others. Worse, ISconf encourages its users to break its own rules; it is unlikely that anyone can read the above examples and think "yeah, gotta follow the rules here." Most sysadmins would prefer to spend the time now to get the new script and the new package management methods working, then replace all of the old instances in specific violation of the principles of ISconf, rather than deal with having cruft in the configuration files for literally years. Having a CMS which has strict rules but encourages its users to break them is self-defeating, and is one of ISconf's biggest liabilities. In fact, this aspect of ISconf is what led me to begin searching for other tools, either to replace or complement ISconf.

Are These Problems Really Intrinsic?

Given that many of the problems discussed in this paper are a result of ISconf's interface to make, it could be argued that these problems are a result of current implementations but are not necessarily intrinsic to ISconf. Unfortunately for those so inclined, replacing make with another tool which better met ISconf's needs could only mitigate the problems, not solve them; ISconf is just a means of associating hosts with the specific commands run on those hosts and the

problem concerns the commands, not the framework. Any attempt at abstracting those specific commands into symbolic information more useful to humans is in direct violiation to ISconf's ordering principles; after all, how can one really verify that work was done in a consistent order if one does not even know exactly what is being done?

A maturity model has been developed for scripts [maturity] that is useful for assessing whether a script will consistently result in a valid, functioning configuration. While it is possible to write from scratch a script that scores highly in this model, it is much safer and easier to rely on proven high-quality tools which fit this model. ISconf's reliance on its users to develop necessary functions makes it less likely that the utilities used will be mature according to the referenced model, which results in a greater likelihood of deployment problems. Other CMS's might not provide the simplicity that ISconf provides, but they are much more likely to have all of their work score highly on the maturity model, which makes for a more stable infrastructure.

As has been mentioned multiple times, ISconf suffers from the fact that it is based entirely on preconditions, but it does not and can not actually code in all of those preconditions. This means that all ISconf stanzas are only valid in the specific environment in which they've been tested, which is created by the sequence of stanzas up to the one in question. Any other use of the stanza is not likely to result in desirable results. The commands executed by the stanza might be independent, but the stanza itself, as run by ISconf, requires the sequence that has already been executed.

Notes on Undecidability and Turing Equivalence

Traugott, et al., recently published a paper [order] claiming that all self-modifying systems, which covers all modern configuration management systems, can be equated to Universal Turing Machines, and thus are susceptible to what is called the Halting Problem [halting], which is a specific instance of the mathematical problem of Undecidability [undecidability]. Traugott, et al., go on to claim that ISconf is actually the only CMS immune to this problem.

While I do not find this comparison particularly worthy of investigation, the claim that ISconf is somehow immune to a problem to which other CMS's are susceptible deserves a closer look. ISconf's theoretical immunity to the Halting Problem arises from its requirement that all actions must be tested by a human before being deployed in production. The implication is that once a given command sequence has been tested in a non-production environment, it can be deployed in a similar production environment and remain immune to any vestiges of the Halting Problem.

Again without addressing the validity of the claim that CMS's are susceptible to the halting problem, if it can be shown that a successfully tested ISconf sequence is still not guaranteed to succeed, then this testing ceases to be any special feature of ISconf, and thus cannot provide any differentiation from other CMS's.

Theoretically speaking, it is trivial to create a command sequence which can succeed during testing but will fail in production: simply build in prerequisites which only exist in the test environment, such as IP addresses, host names, or domain names. Practically speaking, I have seen a number of sequences which succeeded in testing, or even in initial production deployments, which later failed because of differences in pre-existing conditions. I have encountered problems with host installation, package installation, and hostname length, all of which tested just fine but failed in production.

From both theoretical and practical perspectives, it is obvious that testing an ISconf command sequence provides nothing like a guarantee, so this testing can no more protect ISconf from fundamental limitations of the given system than testing with other CMS's can. In fact, because ISconf is so dependent on the bits on the disk and has no higher level facilities for managing objects above the most basic level, it is likely to be more susceptible to fundamental limitations, because it is operating at the same level as those limitations, rather than at a level slightly above them, as psgconf tries to do.

Where to Go From Here

It is apparent that ISconf has fundamental, intrinsic flaws, but the realization that ISconf's rules must be bent allows one to take advantage of what ISconf can do while going elsewhere for more complicated demands. I have found ISconf's ability to map work lists to host names very useful, especially with the typing system as developed in ISconf 3, but found its functionality lacking. Even better, after experimentation with cfengine I found that there were symbolic similarities between ISconf's concept of a host type and cfengine's concept of a host class, and in fact, the format of the two details in the tools was almost exactly the same.

Because I was dissatisfied with the current functionality of ISconf, but wanted to keep the existing work lists, an attempt was made to integrate ISconf with cfengine. Interestingly, significant research effort has been spent in justifying that these two tools are incompatible in their approaches to system management, yet the author found them to be completely orthogonal, and very compatible.

Cfengine Integration

Cfengine is a very useful tool, especially for its its ability to discover the state of the system on which it is running and then perform actions based on that state. However, some actions are obviously intrinsically ordered (one cannot format a volume that has not been created), and ordering arbitrary events in cfengine

is inordinately complicated. Thus, it seemed that cfengine's higher-level capabilities could benefit from ISconf's functionality.

Although I was satisfied with ISconf's ability to perform most work, there was a significant amount of functionality within cfengine that ISconf lacked, such as the ability to verify file permissions and restart processes which died. However, I did not want to just implement cfengine independent of ISconf, because cfengine would need to know how hosts were different and then be able to behave differently according to those differences. Thus, I decided that if I could integrate cfengine and ISconf, such that ISconf had access to all of cfengine's states, and cfengine had access to all of ISconf's host types, I could get the best of both tools without having to store the same information (such as a host being an oracle server) in more than one place.

When this integration project was begun, ISconf used Damian Conway's excellent Parse::RecDescent [recdescent] perl module as a parsing engine, and the format it used for host type names was slightly incompatible with cfengine classes. Because parsing was becoming very slow (taking up to 30 seconds for some key files), the parsing engine was rewritten using Parse::Yapp [yapp] and Parse::Lex [lex], and the format of the host type names was changed to exactly match cfengine classes. This allowed for a simple point of integration: make all ISconf host types available to cfengine as classes, and make all cfengine classes available to ISconf as host types. Thus, attaching a work list to an oracle_server type in ISconf would allow one to set file permissions based on that oracle_server class in cfengine, and discovering in cfengine that a host is a Sun sparc system would allow ISconf to perform specific actions based on that fact.

Unfortunately, the modifications to ISconf's parser were the easiest step in the integration process. Because of a parsing optimization within cfengine, all classes that might be set at some point must be known at parse time, either through hard setting or through use of the AddInstallable command. After much experimentation and many false steps, a script was written that collects all of the ISconf types for a host and generates a cfengine file which sets those types; if cfengine finds that file, then it includes it, and if it does not find the file, then it exits without doing any work. This way one can guarantee that no work is done without all knowledge being available.

An example configuration (trimmed for clarity) is included in Listing 2.

Existing implementations of ISconf have a preparatory script which refreshes its configuration with the most recent version; integration with cfengine enabled an easy replacement of this external script with cfengine's file transport capabilities, on both the client and server side.

Upon successful integration, there was immediate benefit. I had previously written some simple utilities for managing file permissions and ownerships, but those utilities were lacking in key functionality which cfengine possessed, and certainly did not meet the

```
# cfagent.conf
groups:
    # check to see if the istypes.cf file exists
    istypes = ( IsPlain(${workdir}/inputs/istypes.cf) )

import:
    any::
        ispref.cf # creates the istypes.cf file

    istypes::
        istypes.cf
        main.cf

        isconf.cf
        # import all other files if istypes.cf exists;
        # otherwise, import nothing other than isprep.cf
# ispref.cf
control:

    # create the istypes.cf file, and mark all ISconf
    # types as installable
    AddInstallable = ( ExecResult(${workdir}/bin/istypes) )

# isconf.cf
control:
    actionsequence = ( copy "module:runisconf -cf=${ALLCLASSES}" )

copy:
    ${isconfsource}         dest=/var/isconf
                            r=inf
                            server=${isconfserver}
```

Listing 2: Example configuration.

maturity requirements of most organizations. Integration allowed immediate access to all of cfengine's file permission functions, based entirely on information collected from ISconf; in other words, cfengine was able to immediately replace a separate, less capable script without any extra development effort (although it did take some practice with cfengine, obviously).

As this integration has progressed, an interesting conceptual transformation happened; the ISconf host types conceptually became just *facts*, just as Couch, et al., observed that cfengine states are Prolog Facts [prolog]. Rather than being part of a hierarchical typing system, they were just logically true or false, even if they were set that way based on that hierarchical typing system. This allows one to build work lists as a group, without having to consider that work list as a host type. For instance, consider a lengthy application configuration process, such as is necessary for cfengine: installing three different applications, modification of the /etc/services file, creation of the cfengine key pair, downloading the update.conf file, uploading the public key to the cfengine server, and finally running cfengine the first time to get everything started.

It would make no sense to make this into a host type, because all systems will run it. It is clearly just a related, ordered list of work, not a separate type of server. The precepts of ISconf require that one add each item in this list to all configured host types, or at least add them to a base type, with no indication that the work is related; however, if one considers ISconf types to just be classes, rather than mapping to a server type, then one can very easily configure this as an ordered work list, maybe named cfengine, and then attach it to the necessary host types. This retains the fact that these steps are all related, while still satisfying ISconf's ordering requirements.

Another process that became far easier with an integrated cfengine and ISconf was the testing that ISconf requires so much. To test a new stanza in ISconf, one must do a partial roll-out of that stanza, but one can only do that to entire host types, both development and testing. The method ISconf uses to get around this problem is requiring multiple domains (e.g., test and prod), different fileservers for each domain, and then deployment of the new stanza first on the test fileservers then on the production fileservers. This partial deployment requires that all fileservers be treated differently because they must all have different branches of the ISconf configuration. If two people want to do partial roll-outs of code in two unrelated host types, they basically cannot.

Once integrated with cfengine, however, testing new ISconf stanzas becomes far easier. All test and production hosts should be marked so in either cfengine or ISconf (depending on configuration needs); when it comes time to test a new sequence of work, it can be created as an independent work list, and cfengine can initially enable that work list only for test systems. Once the test is complete, cfengine or isconf can then easily enable it for all hosts. For instance:

```
# in a cfengine config file
test_domain_com::
    AddClasses = ( mytestworklist )
```

If the test list succeeds, then it can be deployed on all hosts either through ISconf, by adding it to another ISconf type, or through cfengine by eliminating the test_domain_com:: portion of the file.

For the future, the author would like to begin replacing some of ISconf's configuration files with cfengine. For instance, instead of having a hard mapping of host names to host types in ISconf's hosts file, the mapping could be maintained within cfengine. This would enable cfengine's logical testing capabilities earlier in the ISconf process, but would also sacrifice ISconf's ability to associate arbitrary variables with hosts and host types (cfengine does not currently have a facility for passing arbitrary variables to its modules or shellcommands).

Conclusion

ISconf is billed as an enterprise configuration management system, capable of managing all aspects of system administration, but it is found instead only to be capable of ordering work done by some other system. This finding does not invalidate ISconf's usefulness, but significantly reduces its scope while at the same time allowing it to be used with other tools which have greater functionality. Most of the existing theory explaining ISconf's usefulness is found wanting, and the purported conflict between tools like ISconf which preach congruence and tools like cfengine which preach convergence is found not to exist. Turning ISconf into a cfengine module is found to make both tools significantly better.

Biography

Luke Kanies graduated from Reed College in 1996 with a Bachelor's degree in Chemistry. He has since been honing his skills at using Unix to do less work through automation and abstraction. He currently runs a consulting company, Reductive, LLC. Reach him via email at luke@madstop.com.

Availability

ISconf 3, by Luke Kanies, is available via the web at http://www.sourceforge.net/projects/isconf. Cfengine, by Mark Burgess, is available via the web at http://www.cfengine.org. PSGConf, by Mark Roth, is available at http://www-dev.cites.uiuc.edu/psgconf/.

References

[bootstrap] Traugott, S. and J. Huddleston, "Bootstrapping an infrastructure," *Proc. LISA XII*, 1998.

[cfengine] Burgess, M. "Cfengine: a site configuration engine," *USENIX Computing Systems*, Vol. 8, Num. 3, 1995.

[decidability] *Decidability*, http://wombat.doc.ic.ac.uk/foldoc/foldoc.cgi?decidability.

[isconf] *ISconf: The Infrastructure Configuration Engine*, http://isconf.org.

[lex] Verdret, P., *Parse::Lex perl module*, http://search.cpan.org/author/PVERD/ParseLex-2.15/.

[make] *Make*, http://www.gnu.org/software/make/|.

[maturity] Couch, A., "An Expectant Chat on Script Maturity," *Proc. LISA XIV*, 2000.

[order] Traugott, S. and L. Brown, "Why Order Matters: Turing Equivalence in Automated Systems Administration," *Proc. LISA XVI*, 2002.

[prolog] Couch, A. and M. Gilfix, "It's Elementary, Dear Watson: Applying Logic Programming To Convergent System Management Processes," *Proc. LISA XIII*, 1999.

[psgconf] Roth, M., *PSGConf*, http://www-dev.cites.uiuc.edu/psgconf/.

[recdescent] Conway, D., *Parse::RecDescent perl module*, http://search.cpan.org/dist/Parse-RecDescent/.

[turing] Turing, A., "On Computable Numbers, with an Application to the Entscheidungsproblem," *Proceedings of the London Mathematical Society*, Series 2, Vol. 32, pp. 230-265, 1936-37.

[yapp] Desarmenien, F., *Parse::Yapp perl module*, http://search.cpan.org/author/FDESAR/Parse-Yapp-1.05/.

Seeking Closure in an Open World: A Behavioral Agent Approach to Configuration Management

Alva Couch, John Hart, Elizabeth G. Idhaw, and Dominic Kallas – Tufts University

ABSTRACT

We present a new model of configuration management based upon a hierarchy of simple communicating autonomous agents. Each of these agents is responsible for a "closure": a domain of "semantic predictability" in which declarative commands to the agent have a simple, persistent, portable, and documented effect upon subsequent observable behavior. Closures are built bottom-up to form a management hierarchy based upon the pre-existing dependencies between subsystems in a complex system. Closure agents decompose configuration management via a modularity of effect and behavior that promises to eventually lead to self-organizing systems driven entirely by behavioral specifications, where a system's configuration is free of details that have no observable effect upon system behavior.

Introduction

Most system administrators accept that skilled system and network administration involves being a generalist: integrating bits and pieces of intricate and diverse minutiae into the skills to design a system, provide a service, or troubleshoot a problem. The wizards who can perform this integration teach the apprentices who are not yet wizards, and our configuration management tools are built in the image of the wizards, to allow more apprentices to function with less wizards among them. In short, current tools are aimed at teaching humans to manage an inherently complex process, and to embrace and even contribute to that complexity.

We believe that the complexities in a complex system are often illusory. Many are the result of less than thoughtful design, or at least, design not motivated by a goal of decreasing complexity to make systems more manageable. In this paper, we outline a strategy for reducing complexity and intricacy by changing the level of abstraction at which we interact with systems. If we adopt a new mindset and proceed according to a new set of rules, much of the complexity disappears. It is not defaulted or otherwise hidden by clever lingual mechanisms; it is literally gone and need not ever be considered again.

The key to this process is to "close the box" on subsystems that are sufficiently mature and allow them to become self-managing and self-healing in the absence of an administrator. This is a bottom-up process of administrative "practice hardening" in which we build overall system robustness upon a foundation of highly reliable low-level configuration subsystems that are tightly and inextricably coupled with behavior. These subsystems, together with a set of design rules for building interactive networks of subsystems, form a new paradigm for system administration. Almost everyone involved in configuration management is using some or most of these rules; this paper is an attempt to write down all of the known rules in one place.

With a few exceptions, most current configuration management tools function at an inappropriate level of abstraction. Specifications and declarations concern systems and networks, when they should instead document the behavior of *closures* and *conduits* between closures. A "closure" is a "domain of semantic predictability," a structure in which configuration commands or parameter settings have a documented, predictable, and persistent effect upon the externally observable behavior of software and hardware managed by the closure(more precise definitions will be discussed later). Closures are not entities that live just within one machine, but can also span LANs and networks. "Conduits" are methods of communication between closures, by which they can make their needs known to other closures. A conduit can take most any form, from a command-line interface to a custom networking protocol.

The idea of closure is not new. Many closed-source network devices and subsystems already exhibit some form of closure; commands are guaranteed to work and to be free of external effects. Service appliances and network switches are prime examples. We study instead how to create and maintain closure in an otherwise open environment subject to many changes and updates. We seek predictability in an otherwise unpredictable environment: "closure in an open world."

Configuration Management Challenges

Current configuration management systems all suffer from a similar set of problems that arise from the nature of the task. Among these problems, the

most important for our discussion are the problems of *referents*, *unintended consequences*, *hidden preconditions*, *latent variables*, and *incidental complexity*.

The *problem of referents* [11, 18, 19] arises from the complexity of the systems being configured. In a large and complex network, how does one specify how a particular subsystem should behave? This is a matter of referring to the subsystem and its parameters "by name" and assigning values to each parameter. The problem is that any naming scheme complex enough to precisely specify a subsystem is too complex to remember and use effectively. There are many approaches to hiding the problem of referents with clever language structuring [8, 11, 18, 19, 33].

For example, in Distr [11] and Arusha [18, 19], low-level parameter value declarations can be reduced or avoided via parameter defaults specified at a higher level. Environmental acquisition [33] allows parameter values to be inferred from the context in which a host must operate, much as a red automobile typically has red doors. Unfortunately, clever tricks such as value inheritance and environmental acquisition do not eliminate the problem; they simply transform the problem of referents into the equivalent problem of keeping inherited attributes correct in an increasingly complex inheritance scheme. As we will see, the solution is not to refine the solution, but to change the problem.

The *problem of unintended consequences* [15, 17, 40] arises because subsystems are often coupled in unforeseen and even undocumented ways. Very commonly, replacing a dynamic library to repair one application will break another. Similarly, installing one application can edit, e.g., /etc/inetd.conf, so that another application is disabled. We need some way of protecting ourselves from doing things that harm more than help.

Unintended consequences are often the result of *hidden preconditions*. Every management process, whether automated or manual, only works in particular environments. If we forget the environment and conditions under which a process is applicable, the process may have unintended consequences. An ideal process has minimal preconditions, and those preconditions are explicitly defined.

In turn, all hidden preconditions are the result of *latent variables*. A latent variable is a fact about a system that remains unseen until it causes a failure. For example, in a Linux server of about five years ago, the fact that one network card in a Linux server was manufactured by a particular vendor was unimportant, but if one added another identical card, a well-known networking bug would cause packets to be sent on incorrect interfaces. The manufacturer of the initial card is a latent variable that is not perceived to be a problem until it is *expressed* (like a gene) by adding another identical network card.

These problems obscure yet another problem that is not generally acknowledged, but is far more expensive in terms of human effort: the quandary of *incidental complexity*. While our mission as system administrators is to observe and assure particular kinds of system and network behavior, 95% of the information we currently specify (or perhaps inherit and override) during "configuration management" has no impact upon observable behavior.

A system is a graph of multitudes of interdependent minutiae, requiring knowledge of facts such as:

1. Where particular configuration files, programs, and subsystems are stored.
2. Which environment variables affect which applications.
3. Which disks are faster than others.
4. Which machines get which services from which others.

Bothersome facts about this minutiae include that:

1. An effective system administrator must be able to unravel all of it, ergo
2. One must set up patterns that are easy to remember, so
3. It must be consistent from system to system, even though
4. Its initial specification is not only subject to human error, but can also drift over time due to configuration changes.

The result of this culture is that configuration management systems, faced with the *tradition* of human management of irrelevancy, *streamline* this useless management task instead of more appropriately eliminating it entirely.

A pervasive and systemic problem sometimes requires a radical solution. If information is irrelevant to behavior, we leave that information to a better authority than ourselves: an expert system that figures out the various minutiae necessary to assure a behavior. This expert system takes the form of a suite of small and simply constructed configuration engines responsible for particular facets of behavior. These engines communicate with one another through a hierarchy that reflects the primary dependencies between subsystems. They are designed bottom-up and utilized top-down. A human communicates with the top level in order to effect changes at lower levels.

To solve the problem of incidental complexity, these configuration engines split configuration parameters into two distinct sets:

1. *Interior* parameters that are the responsibility of agents, and are read-only at all times (except during troubleshooting).
2. *Exterior* parameters that are the responsibility of the system administrator and are always read-write.

In a human-readable configuration, only exterior parameters matter, which greatly reduces the size of a "configuration." This division also leads to a less-than-obvious solution to the problem of referents. If referents are *behaviors*, instead of *parameters*, the exterior space of referents self-scales to a size a human administrator can easily memorize and internalize.

The agent approach also solves the problems of unintended consequences and latent variables for the *core* of a system, but not overall. Anything that is suitably constrained cannot behave badly, but it is impractical to constrain everything. The highest level of any system might need to remain open and evolving.

If as well, the exterior parameters are controlled by *constraining values* rather than specifying literal values, we have achieved the highest attainable level of abstraction in specification, a strategy first proposed by Burgess [7] and Anderson, et al. [2].

Closures

A "closure" is a programming language term [41] for a name-binding environment in which setting a variable and then reading it always gives the value to which it was set, independent of the settings of other things. In a closure, the meanings of names are independent of one another, unique, and persistent. Perhaps the simplest closure is `{ . . . }` in C. For example,

```
{ int x=4; { int x=1,y=5; } }
```

defines two nested closures, each with a distinct idea of the binding between variable `x` and a storage location. In the outer closure, `x` has the value 4, while in the inner closure, a different binding for `x` has the value 1. After the inner closure and inside the outer one, the symbol `x` briefly refers again to the one with value 4.

"Closure" in a mathematical sense is a property of a mathematical system in which operations within the system do not produce results outside the system, e.g., we say that the integers are "closed under addition" because the sum of any two integers is an integer. Likewise, the integers are not closed under division, because 1/2 is not an integer. As another mathematical meaning of closure, in real analysis, a set of numbers is "closed" if the limit of any sequence of set members is present in the set.

Generally, in a closed system, no matter what one does under the rules, one obtains a result to which the rules still apply.

We apply the term *closure* to system administration in a form that embodies all of these prior meanings.

Principle 1: A *closure* is a subsystem with highly predictable, reliable, and robust behavior in response to configuration changes.

It is a domain of "semantic predictability" in which the behaviors one requests are portrayed exactly as requested and have a precisely predictable effect.

Coming to closure in a relatively static environment is easy; coping with change is the prime adversary of closure. The key to handling change is to base configuration management upon those attributes whose structure changes least frequently. The implementation of services is constantly changing and evolving, but the *nature* of services – what they do and how they behave – changes more slowly, if at all. We thus base our language of configuration upon *observable behavior* rather than parameter settings, where a behavior is observable if one can determine its presence or absence by asking a simple yes/no question. As software is updated and upgraded, overall behavior changes little with each new software revision, while the underlying low-level implementation may require regular and sometimes drastic improvements.

Parts of a Closure

A closure has four parts:

1. A set of conditions that define the environment in which the closure will operate properly. In software engineering terminology (as well as in the theory of program correctness [41]), these are called *preconditions*.
2. A set of *configuration operations* that work predictably when preconditions are met.
3. A set of *conduits* that allow exchange of information with other closures.
4. A map from configuration operations and parameter values to system behaviors, which specifies how each configuration change must affect the behavior of the overall system. In software engineering terminology, these are called *postconditions*.

The effect of a closure is that if one obeys preconditions and uses only closure operations and conduits, then the map from configuration to behavior will remain a definitive description of behavior. If one violates any of these restrictions, the map is no longer guaranteed.

We do not expect that anyone will argue about the value of predictability. The controversial part is that one must limit one's environment and practice to assure that predictability. It is not practical to allow "anything" to be done to any machine – the result is almost never maintainable or reproducible. In particular, one must not even attempt to configure a system other than by its approved conduits; to do so invites disaster.

Again, this concept is not new. Every software or hardware product embodies some kind of closure. If one installs the product as recommended (preconditions), and interacts with it using configuration operations listed in the documentation, it will hopefully behave as documented (postconditions).

Closures Without Agents

So how difficult is it to construct a closure? Practically, it is just a matter of limiting one's operations and certifying their effects individually and in concert. It does not require an agent, and can instead consist entirely of manual practices.

For example, at Tufts we certify only a few baseline configurations for desktop computers, including hardware and software specifications. These configurations form a closure in which the results of common upgrades become predictable. If a user deviates from a baseline by employing custom hardware or software, he moves outside the closure and (according to support rules) receives a lower level of support. If it is

impossible to repair his problems, staff will offer to put him back into the closure ('baselining') but will not debug problems encountered by a user who voluntarily leaves the closure. The closure allows us to provide a relatively high level of service to people within the closure, at the cost of aggressively discouraging people from leaving it. This saves much support time by avoiding costly troubleshooting sessions on systems outside the baseline.

Closure and Open Systems

Closures are not an attempt to "close systems," and the systems in which closures operate remain otherwise open.

Principle 2: IA closure is a highly predictable subsystem of an otherwise open system.

While it is a good thing to be able to extend systems for any use, the act of extending them is often plagued by latent variables. A closure adds sanity to an otherwise open system by protecting a relatively mature subsystem from the effects of open extensibility. We protect the subsystem not by building physical barriers around it, but by agreeing to manage subsystems in a very disciplined and structured way. Closure of whole systems is not always practical. Instead, we locate subsystems that exhibit closure, and agree to leave them alone to perform their appointed tasks.

Again, this is an old idea with a new name. IP/DHCP is an example of such a closure. If we leave it alone and configure it via well-documented procedures, it will assure networking up to the session layer. Applications can leave this function to the operating system, thus ensuring predictability for IP functions. Likewise, network appliances form closures at the service level for file service, web service, proxying, etc.

Closures and Best Practices

There has been much discussion lately about the concept of "best practices" and their role in organizational robustness of service organizations [26, 30].

Principle 3: An ideal (well-designed) closure is an embodiment of best practices that never allows its subsystem to enter an unapproved or invalid state.

The limitations imposed by a closure and its resulting behavior are a standard amenable to validation and verification. Within the standard there is guaranteed predictability and interoperability; outside the standard, anything can happen.

Some closures are simple static preconditions on otherwise open systems. Software for Linux systems is plagued by incompatibilities between the distributions. The Linux Standard Base (LSB) [42] is a closure that limits the structure of the library binding environment for Linux distributions so that vendor software is guaranteed to work. It specifies standards for how distributions locate system files and libraries. Vendor software written to conform to the standards will work properly in any distribution that conforms to the standards.

LSB has the interesting property that its closure properties span distributions; any application that conforms to LSB standards and works properly in any one conforming environment is guaranteed to work in *all* conforming environments. There is no more need for "write once, debug everywhere;" debugging in one conforming environment is sufficient.

Coming to Closure

Though it is possible to buy highly predictable subsystems, a closure is not always something one buys or downloads. It is mostly a change in the way one thinks about the actions one takes in configuring a network. The main content of a closure is "additional rules of practice" that keep one from creating situations in which latent variables can appear. Break these rules, and one no longer has a closure. Follow them, and predictable behavior is assured.

The simplest closure of which we are aware is "Never delete a dynamic library." This discipline assures that you will be free of one latent effect, that of deleting a library that is in use. If one only installs software, and never overrides the contents of a dynamic library, then proper function of most programs is assured. If one ever deletes or replaces a dynamic library, havoc can result [15, 17]. All closures trade something for predictability and robustness. This practice costs one some administrative flexibility and disk space in return for a more stable and predictable system.

Closures take many forms and scales. A network appliance or thin client is a closure at the *system* level: an autonomous component with predictable and immutable behavior. DHCP can be viewed as a closure at the *network* level; it is a domain of predictable behavior in which clients are always assigned reasonable IP addresses. Network closure is also the purpose of Oracle's "StarNet" network middleware layer, among others; in this case the goal is reliable and secure communication with a database management system. A good operating manual often forms a closure at the *human* level: a set of procedures that "always work." All of these are subsystems that are somehow forged to be highly predictable for what they are intended to accomplish, regardless of whatever else is going on in a complex system.

Conduits

A *conduit* is in the context of a closure has the same meaning that it does in middleware. It is an interface between closures that reliably allows information exchange between them, and mediates configuration changes so that mistakes are less likely in either communicating closure.

Conduits can take many forms. Ideal conduits interface between subsystems with no administrative intervention, e.g., the gethostbyname function that interfaces with the domain name service.

An Ideal Closure

Closures vary in the effectiveness with which they eliminate latent effects. Almost everyone has to manage one or more web servers. Let us explore how practice would change if we tried to eliminate all latent effects by building a closure around a web server. This is going to take a little rethinking about how we "configure" web servers, but is both possible and practical.

In forming this closure, however, we will have to violate some very common conventions of user interaction, including shell accounts on the web server. We will do this with the confidence that it closes the system against unpredictable effects in ways that a shell-based system cannot be closed. In creating the closure, we will outline general principles one can apply to any situation.

Taxonomy of Behavior

To achieve a closure that "acts like a web server," we will lift details of the configuration of a web server from its phenomenology as a server, i.e., the set of behaviors that make it a web server. The goal will be to describe the server from the outside, and allow the server to configure its internals without human intervention.

Principle 4: An ideal closure's configuration language is derived from a taxonomy of desired behaviors, not from the internal taxonomy of the system.

What comprises a web server's behavior? It has an IP address, responds to one or more names and/or ports, delivers content for each URL one provides, calls CGIs, and perhaps calls extension modules or interacts with database servers. So its configuration attributes should include:

1. *Identity:* IP addresses, virtual names, certificates, ports on which to listen.
2. *Customization:* Names of required modules, libraries, database bindings.
3. *Content:* A map from (virtual name, port) to a hierarchy of files, including service constraints.
4. *Auditing:* Ability to retrieve descriptions of service activity.

For now, let us consider this an exhaustive list of *everything* we provide to the server or get back from it. Everything else is going to be accomplished automatically without intervention from us.

Our pattern-slaved nature as human beings leads us to unavoidably fill in details where they are not needed, so it is difficult at first glance to see just how much we just left out of the web server's configuration. We left out:

1. The operating system to use.
2. The layout of the operating system.
3. The web server to use.
4. The representation of data.
5. The specific locations of files, including source, object, libraries, modules, etc.
6. Performance tuning.
7. Everything the web server requires of its environment that is not seen by the user.

Obviously these details must be "added back in" at some later time. Ideally the closure itself provides all these details during its efforts to install itself. We thus minimize what the administrator has to specify and learn.

There is not really any need for an administrator to know where files are kept on a functioning web server. This information is only useful if the server fails and one must troubleshoot it. If closure can arrange for it never to fail or to be self-correcting, then information on its internal structure is no longer needed.

Isolation

Our first configuration action will be to remove normal users and normal configuration tools from the system that will be a web server:

Principle 5: A closure is isolated from subsystems that might create latent effects. ed As part of our contract with the closure, we will "let it manage itself," so we do not need the ability to manage it or interact with it otherwise. This is good, because:

Principle 6: An ideal closure's internal structure is completely opaque to the user; it can vary with circumstances and utilize the most efficient internal representation for a specific environment.

An excellent example of self-optimizing behavior may be found in [2].

Our web closure will presume that it is the sole manager of its configuration. Any violation of that contract will seriously affect our closure's ability to manage itself, because there will be latent effects of changes that the closure did not make during self-management. Our closure's ability to repair itself is greatly tempered by being able to control what changes occur and when. If it has complete control of its configuration, this is a matter of simple feedback algorithms, while if it did not have total control, it would instead be forced to rely on pattern matching and machine learning. We wish to avoid this due to the typical unpredictability of such control mechanisms.

In our particular case, data will be stored in the system in the way that's most efficient for the system, and need not reflect external architecture unless necessary for some internal reason. The reasons for this are obvious; the user and the storage medium are unnecessary constraints that can lower the efficiency with which we can maintain the closure. By removing these constraints, we also increase the size of the space of alternatives for implementing a solution. In particular, we can supply web pages in HTML while the closure stores them in XML.

Conduits

Conduits are the *sole* interface between the administrator and the closure.

Principle 7: A closure is self-centered and requires one to communicate all configuration changes and gather performance data through a gatekeeper conduit.

The gatekeeper conduit is the mediator during contractual disputes between administrator and closure. If you ask a closure to do something it can't do, the gatekeeper will reject it:

Principle 8: A closure is self-policing and validates configuration changes before making them.

This enforces *integrity constraints* that keep the closure configured in a proven way. The key to successful closure is that *every* achievable configuration corresponds to one of a set of *best practices* [26, 30], so that undesirable states are never entered.

The gatekeeper conduit may be a human being or software. The conditions on what a gatekeeper (of any kind) will accept must be carefully documented. Every closure's documentation must describe four things:

1. Prerequisites for setting up a closure, including hardware, software, network resources, etc.
2. Techniques for interacting with the closure as a holistic entity.
3. Postconditions and expectations: what will happen if you do this.
4. Consequences of violating preconditions: what will happen if something breaks.

Documentation is particularly important since it is our *only* clue to how a closure will behave in a specific situation.

For web services, there are at least three kinds of required gatekeeper conduits.

1. Configuration: Determines how the server behaves overall. This can be any kind of user interface that describes behavior other than content.
2. Content: For each virtual service, a conduit that describes data to be served.
3. Auditing: Describes log content.

Configuration is done using nothing more than the usual GUI or CLI (or preferably both) that describes what virtual servers to implement and what special behavior each should exhibit.

The content conduit is dramatically different. It is a closed interface to providing content that can support one or more of:

1. One-way or two-way mirroring of a filesystem available to developers.
2. An explicit CLI or GUI for creating content.
3. A database feed.

This roundabout way of providing content may seem bizarre until one realizes that we are attempting to completely eliminate operational coupling between the closure and the outside world, except through conduits, and arrange for robust service even if conduits are interrupted. Many sites already configure their web servers this way for precisely the same reasons, but call the practice "content staging."

Employing a conduit for content also allows the closure to utilize its own representation for the content that is independent of that provided by users. One thing that one does *not* want to do is to give the closure direct access to the files being updated by users. This creates points of failure that do not exist if these files are filtered through a conduit instead. For example, there is no way that a file provided through a conduit can have an invalid file protection, but a human manually arranging to publish a file can easily mis-protect the file in a variety of ways.

Environment

Few closures can be created in a vacuum. They must take information from the network in order to initially configure themselves, either from the Internet or from a specific host designated as a fileserver. They are also frightfully dependent upon having the resources they expect for disk space, memory, and network bandwidth. All we can do is to limit a closure's dependence upon the outside world in some strategic way:

Principle 9: An ideal closure is self-contained; it only depends upon external resources during configuration and is otherwise uncoupled from its network environment.

This eliminates many latent effects of changes in the environment, such as reboots of other servers, etc. A closure is like a binding contract: get this, do this, and this will happen. There are negative consequences as well as positive ones. Our web server will be very unhappy if it is memory-starved, and non-functional if it is disk-starved in storing content.

Preconditions

For our web server, let us presume that we document the following requirements:

1. 128 MB or more of main memory.
2. 10 GB of available disk.
3. Intel pentium 300 or above.
4. Platform in the RedHat certified list.
5. Extra hardware in the RedHat certified devices list.
6. Initially connected to a network supporting DHCP, with access to an appropriate RedHat repository.

This is more or less all we need to arrange an automatic build of a webserver with no human intervention.

Awareness

Principle 10: An ideal closure is responsible for assuring the integrity of its operating environment.

Documentation of the closure will state specific hardware and resource requirements, and service software will check for those requirements on an ongoing basis and fail otherwise. The reason is obvious; the closure is the best judge of what is needed for it to function. These constraints could be as general as ensuring hardware function or as specific as checking for integrity constraints in the operating environment. If it cannot change its environment to suit, the closure will request service from humans. You must then "adapt to service it."

In the case of our web server, the checks the closure should make are easy to construct. One can arrange

to check for available memory, disk, and devices at boot time, and fail if appropriate devices are not present. The ideal closure assesses its environment and creates its initial configuration based upon what it finds.

Principle 11: An ideal closure is self-organizing and self-installing.

An ideal closure could build itself in entirety on bare metal from a floppy disk and network. This is not at all unreasonable given the current state of the art, though it does limit one's operating system choices. We are aware of many people who have developed boot floppies for various kinds of services, notably cache servers, routers, firewalls, and thin clients.

Security

It is past due for our software to take some of the responsibility for its own security.

Principle 12: An ideal closure is self-hardening and self-repairing, and patches itself for security problems when information on the problems becomes available.

When an administrator decides that a patch is warranted, the closure itself installs it. This seems scary but is relatively easy to arrange, because the closure already certified the environment in which the closure operates. So we know whether a patch will install correctly if it installs correctly on another host running the same closure.

But the state of current monitoring and response technology goes far beyond simple patching. One can easily construct a web service environment in which available of service is constantly tested and the service environment is restarted whenever it crashes. In many cases, restarts are not emergencies and will simply appear in the administrator's normal log of activities for a server.

Adaptability

A closure is responsible for any flexibility one has in modifying hardware over time.

Principle 13: An ideal closure is self-reliant and deals internally with time-varying hardware changes and resource availability.

This could range from no flexibility, meaning that you can't change anything without a bare-metal rebuild, to total flexibility, that arranges to deal elegantly with complete machine replacement by periodically backing up servers and cloning them on other hardware.

Caveats

There are several invariants of closures that may not be obvious at first glance.

1. Closures are not necessarily portable. They depend upon a contract more specific than to run on any hardware whatever. They can be customized to a specific environment to make a *new* closure.
2. The contract for a closure is a homogeneity constraint. It assures that every closure starts building itself from the same basic set of resources. Closures are scalable provided that homogeneity constraints are satisfied system-wide, and unscalable otherwise. A closure can be replicated without bound once we know how to build systems on which the closure can live.
3. Closures reduce what one has to watch about a network and reduce possible troubleshooting causes. Something else other than the administrator is watching the closure. If anything goes wrong with the closure itself, it will be caught by the closure's own management mechanisms.

The above example is extreme, and much milder examples of closures can be created, e.g., we can make a web server that co-exists with user services by creating a different contract. We can create a closure for IP Telephony by, e.g., standardizing sound cards and drivers and creating an installer and maintenance engine driven by a central knowledge base.

Some seemingly obvious facts about closures are false. The union of several closures need not be a closure. Consider two closures:

1. rpm --install commands on an rpm-compliant host with matching rpm repository.
2. make install in source directories.

Both of these commands are closures in the sense that there is a sequence of appropriate operations that gives any desired effect in each closed world.

But the union of these two closures is seldom a closure. The problem comes from the way they interact when a make install (using autoconf) binds to a dynamic library provided by a distinct rpm -install. The rpm closure, which has inverses when considered as a closed world, no longer has inverses when considered as part of the make install closure. If this library is removed, the result of the make install will break. So the rpm command that installed the library cannot be undone without breaking the make install closure.

A Model of Closure

Further design principles require a precise definition of closures. This section is difficult reading and can be skipped without much loss of continuity. Here we precisely define the concept of closure and mathematically illustrate some more esoteric principles of closure design.

Our concept of closure will be a property of a set of software agents. An *agent* is an autonomous process, within a system or network, that accomplishes changes based upon external commands from a system administrator or other agents. Agents can be embodied in physical processes or combined into a single process with multiple functions, as in Stem [25].

In designing a closure, there are a few inescapable and well-justified principles that limit our design choices. To start,

Axiom 1: A usable set of agents must be convergent, consistent, aware, and atomic [14].

1. A *convergent agent* is defined as a software process that will do nothing unless something is amiss, and will correct anything within its domain of change that becomes non-compliant. This property is necessary to limit the intrusiveness of using an agent, and limit downtime due to agent actions. "If it ain't broke, don't fix it."
2. A set of agents is called *consistent* if agents will not undo other agents' actions. This we formerly called *homogeneity* [14]. This is necessary so that a set of agents will, at a particular time, agree upon a state to assert for a system.
3. An agent is called *aware* if it knows whether an error occurred in making a change. This is necessary in order to ensure atomicity, below.
4. In database theory, an action is *atomic* if it either succeeds and accomplishes a change, or fails with no change whatever. Similarly, an *atomic* agent leaves a system completely unchanged when an error occurs during a configuration change.

Convergence and consistency assure that cooperating agents will not interfere with one another or the user. Awareness and atomicity preserve integrity of the underlying system in case of configuration failures, and assures that if the system cannot be forced into compliance with the ideal, that the act of enforcement does not create further problems. In other words, "a closure should do no harm."

Ideal agents treat the process of configuration as if it consists of database transactions [22, 23], where allowable transactions include constraints as well as literal assertions of state. These transactions are subject to integrity constraints and will fail and do nothing if a desired state cannot be achieved.

Alas, the above obvious properties of an agent are not enough to allow us to construct a system management environment from cooperating agents. Informally, we say that "each agent manages a closure." The exact and precise meaning of this takes some work to develop. One deep problem concerns the meaning of *consistency* between agents. The most challenging problem is to design closures so that local consistency between pairs of closures is sufficient to assure global consistency of closures as a set. Pairwise consistency requires up to $O(n^2)$ operations to verify, where n is the number of closures, while global consistency may require up to $O(2^n)$ consistency checks, one for each subset. The rest of this section discusses how to avoid the latter expensive process of consistency checking.

Parameters and Configurations

In understanding the mathematics of closures, we must first define the notion of what a closure does. In the following, subsystems and closures (a special kind of subsystem) are notated in capitals, while their attributes are notated as script capitals subscripted with the subsystem to which they apply, e.g, $\mathcal{V}_A$.

Definition 1: For each *subsystem* A, let $\mathcal{V}_A$ be the set of its configuration parameters.

'V' stands for *variant*. We purposefully leave the concept of parameter relatively unconstrained. To handle complex situations, some parameters may be "latent" or "unexpressed," like "unexpressed genes." For example, in a web server, theoretically every directory on the system has an access list, but only the access lists of directories that serve as web content are expressed by having an impact upon operation. So a subsystem may have an infinite number of possible parameters, though only a finite number are expressed at any time by actually controlling behavior.

We next need to allow parameters to have values.

Axiom 2: Without loss of generality, we can consider each configuration parameter value to be a string from some fixed alphabet Σ.

Since a string can contain any structure, including XML, arbitrary hierarchical relationships can be portrayed.

Definition 2: A configuration c of A is a mapping

$$c : \mathcal{V}_A \rightarrow \Sigma^*$$

where Σ is an alphabet of configuration symbols and Σ^* is the set of all words (i.e., sequences of characters) that can be formed from the alphabet Σ.

Configurations are assertions that indicate what should be true of a system. They are ideals; the actual parameter values on the system may differ due to external influences. For $p \in \mathcal{V}_A$, $c(p)$ represents its value in the configuration c.

Definition 3: For a subsystem A, let $\mathcal{U}_A$ denote the set of all possible configurations.

'U' stands for *universal*. This is a very large set that, while actually finite due to memory bounds, might as well be infinite. Fortunately, we can limit ourselves to studying a reasonably small subset.

Definition 4: For a subsystem A, let $\mathcal{D}_A \subset \mathcal{U}_A$ be the set of all *reasonable* configurations of A. These are configurations that, according to some standard of practice, make sense.

'D' stands for *domain*. These configurations comprise the domain of change that an an agent can understand and manipulate. The contents of $\mathcal{D}_A$ are a set of choice configurations that exhibit appropriate constraints and behaviors. The structure of these constraints will be a design choice in building an agent.

At any particular time t, a subsystem A exhibits exactly one configuration $c_{A,t}$ that may or may not be a member of $\mathcal{D}_A$. In this paper, however, we will study properties of configurations that do not depend upon time, so that our model will ignore the time-varying nature of configuration and concentrate instead upon static structure. In the argument that follows, all choices we make apply to a single time step. One excellent time-varying model of configuration may be studied in [7]; our time-varying model (which differs somewhat) is left as future work.

Policies

The following is borrowed from the work of Burgess [7] and Anderson [2] but used in a new way.

Definition 5: For each subsystem A (at the time step that we consider), a *policy* is a subset $\mathcal{Q}_A \subseteq \mathcal{D}_A$.

The intent of a policy is to describe a set of options for configuring a system. Each element of a policy describes a reasonable configuration that can be applied to a system. Ideally, the options in a policy result in similar behaviors when applied to the system being configured. Like configurations, policies are ideals. They describe what should be set in the configuration of a machine, not what actually appears there due to the effects of time and change.

Definition 6: A subsystem A's configuration is *compliant with a policy* $\mathcal{Q}_A$ (in the time step that we consider) if the actual configuration $c_A \in \mathcal{Q}_A$.

If a system is compliant with a policy, all parameter values specified in one configuration of the policy are echoed in the actual configuration of the machine.

Definition 7: For a particular subsystem A, let $\tilde{\mathcal{P}}_A$ represent the set of all reasonable policies for A.

A reasonable policy is one that has been implemented, validated, and incorporated into a site practice manual. $\tilde{\mathcal{P}}_A$ is a set of sets of configurations. Each $\mathcal{Q}_A \in \tilde{\mathcal{P}}_A$ is a set of configurations, while each configuration $c \in \mathcal{Q}_A$ is a reasonable configuration in $\mathcal{D}_A$. Again, this is not the set of all *possible* policies, but rather the set of all *reasonable* ones that we think will have acceptable results. The former set is staggeringly huge; the latter is no larger than the detail of one's practice manual.

Policies are "constraint spaces" in the sense of Burgess [6, 7] or Anderson [2]. A policy is not a definition of what must happen, but rather a list of reasonable options. In the current model, for simplicity, there are no priorities or weights for policy options; any option in the list is fine.

We realize that this is a misuse of the word *policy* (which many authors believe to be "that which is determined by management") but at least, it is a consistent misuse among *this* paper's references!

Behavior

One new idea of this paper is to utilize *testing* and *validation* as the definition of external behavior. This continues the work started in [15, 17].

Definition 8: A behavioral test is a yes/no question that determines whether or not a configuration has a particular property. For each subsystem A, let $\mathcal{T}_A$ be a set of behavioral tests that characterize its external behavior.

These are yes/no questions that determine whether a configuration has a particular property or not. Sample tests might include:

- Does A run a web server on port 80?
- Does A not answer tftp requests?
- Does A have a directory named /var/local?

Questions do not have parameters; "Does A run a web server on port 8080?" is a different question than the one above.

Closures

Now we are ready to define the exact nature of a closure.

Definition 9: A *closure* is a quintuple $(A, \mathcal{V}_A, \mathcal{D}_A, \tilde{\mathcal{P}}_A, \mathcal{T}_A)$ such that for any policy $\mathcal{Q} \in \tilde{\mathcal{P}}_A$ $(\mathcal{Q} \subset \mathcal{D}_A)$, any configuration $c \in \mathcal{Q}$, and any test $f \in \mathcal{T}_A$, the value of $f(c)$ does not depend upon the particular choice of $c \in \mathcal{Q}$.

In other words, policy unambiguously determines behavior. To ease notation, we refer to the quintuple $(A, \mathcal{V}_A, \mathcal{D}_A, \tilde{\mathcal{P}}_A, \mathcal{T}_A)$ as the closure A.

Note that the idea of closure depends very much upon what we value (or wish to avoid) through testing:

Principle 14: Designing a closure requires choosing first exactly which kinds of behaviors the closure should be able to produce by configuring a system.

This is embodied in the tests $\mathcal{T}_A$ that determine, among other things, which behaviors are important and which are frivolous. Important behaviors correspond to tests; incidental or frivolous behaviors are those for which there is no test included.

Note that this is as much a condition on the structure of reasonable policies as it is upon their effects. For example, if there is one and only one reasonable policy, and tests always have the same results, we have an administratively trivial closure for which there is only one behavioral result. This means that:

Principle 15: Designing a closure requires carefully delimiting what is reasonable and appropriate, as integrity constraints on parameter space.

In most systems, randomly choosing values for parameters results in chaos. The choices for reasonable configurations $\mathcal{D}_A$ and reasonable policies $\tilde{\mathcal{P}}_A$ exclude such chaos.

If we know A is a closure, then there are maps from configuration and policy to behavior.

Definition 10: For a closure A (which is the quintuple $(A, \mathcal{V}_A, \mathcal{D}_A, \tilde{\mathcal{P}}_A, \mathcal{T}_A)$), let τ_A denote a map from configurations to behaviors

$$\tau_A : \mathcal{D}_A \rightarrow (\mathcal{T}_A \times \{\text{true}, \text{false}\})$$

such that for each configuration $c \in \mathcal{D}_A$,

$$\tau_A(c) = \{(f, f(c)) \mid f \in \mathcal{T}_A\}$$

where $f(c)$ is the (boolean) result of applying test f to a system compliant with the configuration c.

The action of the test suite, $\mathcal{T}_A$, on a particular configuration is to tabulate the results of running all tests, filed by test name and result. The result is a set of ordered pairs $\tau_A(c)$. Because A is a closure, each $f \in \mathcal{T}_A$ can appear in exactly one ordered pair, $(f, f(c))$.

Generalizing this to policies that are sets of configurations:

Definition 11: For a closure A, let $\tilde{\tau}_A$ denote a map from policies to behaviors, where

$$\tilde{\tau}_A : \tilde{\mathcal{P}}_A \rightarrow (\mathcal{T}_A \times \{\text{true}, \text{false}\})$$

such that for each policy $\mathcal{Q} \in \tilde{\mathcal{P}}_A$,

$$\tilde{\tau}_A(\mathcal{Q}) = \{(f, f(c)) \mid f \in \mathcal{T}_A, c \in \mathcal{Q}\}$$

where $f(c)$ again represents the (boolean) result of applying test f to a system compliant with the configuration c.

As before (since A is a closure), each $f \in \mathcal{T}_A$ appears in exactly one pair, $(f, f(\mathcal{Q}))$, in $\tilde{\tau}_A(\mathcal{Q})$.

Note that both of these definitions are nonsense for non-closures, because in the absence of a closure, the results $\tau_A(c)$ and $\tilde{\tau}_A(\mathcal{Q})$ may differ over time even for constant choices of c and $\mathcal{Q}$! By the definition, A is a closure if and only if $\tilde{\tau}_A$ makes sense as a function.

Provided that A is a closure, the contents of $\tilde{\mathcal{P}}_A$ are constrained. Each element of $\tilde{\mathcal{P}}_A$ must be a subset of some *inverse image* of a unique set of behavioral test results, e.g., for a set of test results $L \subset (\mathcal{T}_A \times \{\text{true}, \text{false}\})$,

$$\tau_A^{-1}(L) = \{c \in \mathcal{D}_A \mid \tau_A(c) = L\}$$

Note that for every valid set of test results $L \subset (\mathcal{T}_A \times \{\text{true}, \text{false}\})$, each test appears only once on the left-hand side.

Definition 12: Let $\mathcal{L}_A$ be the set of all possible test results, i.e., all subsets of $(\mathcal{T}_A \times \text{true}, \text{false})$ where each first coordinate appears exactly once.

Then the inverse images of $L \in \mathcal{L}_A$ under τ_A partition $\mathcal{D}_A$ into disjoint subsets, one per non-empty inverse image. Each element of the partition is a candidate policy for the closure, i.e., a set of configurations with identical behavior. In choosing $\tilde{\mathcal{P}}_A$, for A to be a closure, each element of $\tilde{\mathcal{P}}_A$ must be a subset of one inverse image; otherwise policy would not uniquely determine behavior.

Since multiple policies with the same behavior represent wasted effort, without loss of generality we can set

$$\tilde{\mathcal{P}}_A = \{\, \tau_A^{-1}(L) \mid L \in \mathcal{L}_A \,\}$$

This is the set of all subsets of configurations corresponding to unique observable behaviors $L \in \mathcal{L}_A$. Thus

Principle 16: The structure of parameter space for a closure is dependent upon the limited taxonomy of a finite number of *desirable* behaviors, not upon the *possible* behaviors that result from arbitrary parameter settings.

The space of desirable parameter combinations $\tilde{\mathcal{P}}_A$ for a closure A is almost always much smaller than the set of all possible parameter values. It is instead a compendium of best practices for the target environment [26, 30]. This expertise factor makes the inverse function practical to enumerate by iterating over and testing all policy choices. Otherwise, determining the inverse would be intractable.

It is particularly important to limit predictability to a finite subset of objective tests. Many mistakes in reasoning have been made by presuming that behavior includes "all" that a system can do. The observable behaviors are a finite set, while "all behaviors" include minutiae that are not important, e.g., the host's specific MAC address. By constraining the tests, we remove tests that have no impact upon usability. Failing to constrain tests to a finite set is the root of several posed theoretical problems of system administration [20, 38].

Consistency

One of our tenets is that at any time, the set of all closures operating upon a system must be "consistent." We have not yet precisely determined what consistency means. If the closures have disjoint parameter spaces, then they are trivially consistent because conflicts are impossible. Consistency is only nontrivial when two closures share a resource or parameter. The exact nature of that sharing is yet to be determined, and there is a danger of over-limiting closures so that they become impractical to construct. We must define consistency for configurations, policies, and closures.

Configurations are consistent if they agree upon the values of common parameters.

Definition 13: Two configurations $c \in \mathcal{D}_A$, $c' \in \mathcal{D}_B$ are *consistent* if for every parameter $p \in \mathcal{V}_A \cap \mathcal{V}_B$, $c(p) = c'(p)$.

Definition 14: A collection $\mathcal{M}$ of configurations is consistent if any pair of configurations $c, c' \in \mathcal{M}$ are consistent.

In this case, consistency of pairs is sufficient to assure consistency of arbitrary subsets. This is not true in general.

Consistency of two policies means that the policies can both be applied at the same time without conflict:

Definition 15: For closures A and B with policies $\mathcal{Q}_A$ and $\mathcal{Q}_B$, policy $\mathcal{Q}_A$ *admits* $\mathcal{Q}_B$ ($\mathcal{Q} \triangleright \mathcal{Q}_B$) if for every $c \in \mathcal{Q}_A$, there is a $c' \in \mathcal{Q}_B$ such that $c(x) = c'(x)$ for all $x \in \mathcal{V}_A \cap \mathcal{V}_B$.

In words, $\mathcal{Q}_A$ admits $\mathcal{Q}_B$ if for all configurations of A compliant with A's policy $\mathcal{Q}_A$, there is a matching configuration of B compliant with B's policy $\mathcal{Q}_B$. If every possible configuration of either A or B is compatible with some configuration of the other, we have consistency of policies:

Definition 16: Policies $\mathcal{Q}_A$ and $\mathcal{Q}_B$ are *consistent* if $\mathcal{Q}_A$ admits $\mathcal{Q}_B$ and $\mathcal{Q}_B$ admits $\mathcal{Q}_A$ ($\mathcal{Q}_A \triangleright \mathcal{Q}_B$ and $\mathcal{Q}_B \triangleright \mathcal{Q}_A$).

Consistency of two closures means that the policies for the two agents are coupled, so that on common parameters, a reasonable state for one is also reasonable for the other. If the agent for one policy is invoked before an agent for another consistent policy, the second agent will not change parameters that the first agent already corrected, unless these have in the meantime been changed again by an outside force.

The mystery here is how to choose policies for a large set of closures that will be mutually consistent. Informally, a set of closures $\mathcal{S}$ is *mutually* consistent if for any subset

$$\mathcal{I} = \{I_1, \ldots, I_k\} \subseteq S$$

and any consistent choice for policies $\mathcal{Q}_j \in \tilde{\mathcal{P}}_{I_j}$, there are policies for the remainder of $\mathcal{S}$ that remain consistent with this initial set of policy choices. This definition, however, is too naive; closures are not created equal. Some are more likely to be controlled by administrators than others, so that mutual consistency must be based upon choosing policies for those distinguished closures first.

Dominance

Some closures are more important than others in describing consistency.

Definition 17: Closure A dominates B (written $A \succ B$ or $B \prec A$) if for each policy $\mathcal{Q}_A \in \tilde{\mathcal{P}}_A$, there is at least one policy $\mathcal{Q}_B \in \tilde{\mathcal{P}}_B$, such that $\mathcal{Q}_A$ admits $\mathcal{Q}_B$ ($\mathcal{Q}_A \triangleright \mathcal{Q}_B$).

A dominates B if it is possible to bring B into compliance with A by choice of some policy for B. While consistency is a property of a configuration, dominance is a property of the sets of all policies: a constraint on the globally achievable states for both A and B.

Definition 18: Closures A and B are *mutually dominant* if A dominates B and B dominates A.

This is common when the closures are tightly coupled, e.g., DNS and DHCP. Putting a host into DHCP *should* correspond with putting it into DNS and vice-versa.

Some seemingly obvious statements about dominance are false in general.

Proposition 1: Dominance is not necessarily transitive ($C \succ B$ and $B \succ A$ does not necessarily mean that $C \succ A$).

Proof: Construct closures A, B, and C with $\mathcal{V}_A = \{x, y\}$, $\mathcal{V}_B = \{y, z\}$, and $\mathcal{V}_C = \{x, z\}$, and let the policies be assigned as follows (also see Figure 1):

$$\tilde{\mathcal{P}}_A = \{x = 0, y = 0\}$$
$$\tilde{\mathcal{P}}_B = \{y = 0, z = 0\}$$
$$\tilde{\mathcal{P}}_C = \{x = 1, z = 0\}$$

Then $C \succ B$ ($x = 1, y = z = 0$) and $B \succ A$ ($x = y = z = 0$), but $C \not\succ A$ (x values incompatible). □

Dominance is important because it allows behavior to be developed in a stepwise process. If $A \succ B$, then choosing a behavior (via a policy) for A partially determines the allowable behaviors for B, with the remaining options determined by the nature of B or the administrator.

Consistency of Closures

Now we are ready to define consistency of closures. First we need to distinguish between constrained and unconstrained closures in an arbitrary set of closures $\mathcal{S}$:

Definition 19: In a set of closures $\mathcal{S}$, a closure $A \in \mathcal{S}$ is *exterior* if everything that dominates it is mutually dominant with it, and *interior* otherwise.

An exterior closure is one whose parameter values control the parameter values of other closures.

Definition 20: An *exterior cover* for a set of closures $\mathcal{S}$ is a subset $\mathcal{I} \subseteq \mathcal{S}$ where each $u \in \mathcal{I}$ is exterior, every element $s \in \mathcal{S}$ is dominated by some $u \in \mathcal{I}$, and there are no mutually dominant pairs of closures in $\mathcal{I}$.

'I' stands for *interface*. An exterior cover is a set of closures that, once configured, determine the configuration of every closure (though perhaps not uniquely). First, an exterior closure must be "at the highest level" of a dominance hierarchy, with nothing dominating any closure in it that is not mutually dominant with the closure. We require that every element of $\mathcal{S}$ is dominated by something in $\mathcal{I}$, so that the configuration of every element is thereby constrained. A mutually dominant pair is redundant; only one of the pair need be included in order to control both elements of the pair. There are often many choices for an exterior cover due to mutual dominance.

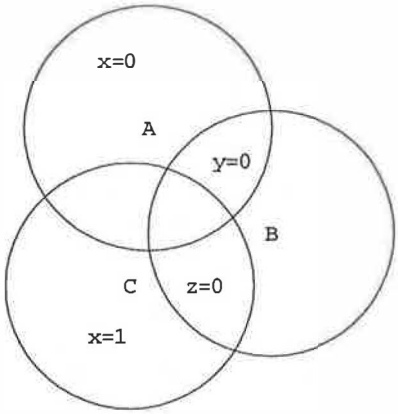


Figure 1: How transitivity of dominance fails.

An exterior cover for a set of closures is a choice for how a human might communicate desires to a set of communicating closures. The exterior closures are the ones with which a human communicates; the interior ones are all implicitly configured by settings for the exterior ones.

We can now characterize what it means for a set of closures to be consistent:

Definition 21: A set of closures S is *consistent* if, given any choice of policies for any exterior cover $\mathcal{I} \subset \mathcal{S}$, there are choices for policies for all non-exterior closures in $\mathcal{S} - \mathcal{I}$ (where $-$ indicates set difference), as well as choices of particular configurations within each policy, so that the set of all resulting configurations is consistent.

Remember that dominance allows us to determine how to make a *pair* of closures consistent. Consistency of a *set* of closures is a much stronger criterion

than pairwise dominance, and pairwise dominance is not enough to assure consistency. In particular,

Proposition 2: A set $\mathcal{S}$ of closures in which for each A and B, either $A \succ B$ or $B \succ A$, and for which $\succ$ is transitive, is *not* necessarily consistent!

Proof: Construct closures A, B, and C with $\mathcal{V}_A = \{x, z\}$, $\mathcal{V}_B = \{y, z\}$, and $\mathcal{V}_C = \{x, y, z\}$ and let the policies be assigned as follows:

$$\tilde{\mathcal{P}}_A = \{\ \{x = 0, z = 0\},\ \{x = 1, z = 1\}\ \}$$
$$\tilde{\mathcal{P}}_B = \{\ \{y = 0, z = 1\},\ \{y = 1, z = 0\}\ \}$$
$$\tilde{\mathcal{P}}_C = \{\ \{x = y = 0, z = 0\},\ \{x = y = 1, z = 1\},\ \{x = y = 0, z = 1\},\ \{x = y = 1, z = 0\}\ \}$$

Transitivity of dominance between these closures is obvious from inspection. A and B form an exterior cover, but there is no policy *globally* consistent with $x = 0, z = 0$ in $\tilde{\mathcal{P}}_A$, because y must be 1 in $\tilde{\mathcal{P}}_B$ and 0 in $\tilde{\mathcal{P}}_C$. By the structure of A, B, and C, there are two conflicting assignments for the same parameter y. □

But, with a few more conditions, we can assure global consistency of closures.

Proposition 3: Let $\mathcal{S}$ be a set of closures such that:

1. For each A, $B \in \mathcal{S}$ where $\mathcal{V}_A \cap \mathcal{V}_B$ is non-empty, either $A \succ B$ or $B \succ A$.
2. If $A \succ B$, then $\mathcal{V}_A \supseteq \mathcal{V}_B$.

Then $\mathcal{S}$ is consistent.

Proof: Note first that in this case, for closures A, B, C, if $A \succ B \succ C$, then $\mathcal{V}_A \supseteq \mathcal{V}_B \supseteq \mathcal{V}_C$, so that by dominance, values of $\mathcal{V}_A$ determine the values in $\mathcal{V}_C$. Thus $A \succ C$, and $\succ$ is transitive.

Note also that any conforming set of closures has the structure of a forest of disjoint trees, as illustrated in Figure 2. Dominant closures are ancestors of dominated closures, where a closure is dominant whenever it contains more parameters than another.

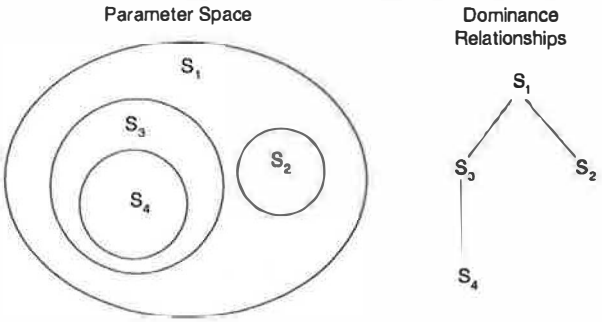


Figure 2: Subset relationships in parameter space exhibit a tree structure.

If $\mathcal{S}$ contains one closure, we are done. If $\mathcal{S}$ contains two closures, then either their parameter sets are disjoint or one closure dominates the other. In the former case, we are done, because the two closures agree on an empty set of common parameters. In the latter case, the dominating closure contains the parameter space of the other, so by definition of dominance, the closures agree on parameter values and are consistent.

If $\mathcal{S}$ contains three closures A, B, C, then if any one has a set of parameters that is disjoint from the other parameter sets, then we have only two closures with common parameters, and we are done by the previous argument. So presume that all three parameter sets have elements in common. By enumerating the possibilities for dominance, one of the three closures must always contain all parameters of the others and dominate the other two, so we are done.

Now assume that the proposition is true for n closures, and consider $n + 1$ closures $\{S_1, S_2, \ldots, S_{n+1}\}$. This set must contain at least one interior closure S_k that is mutually dominant with anything that it dominates; this S_k is a lower bound for the finite set of closures under $\succ$. By induction, the theorem is true for

$$\{S_1, S_2, \ldots, S_{k-1}, S_{k+1}, \ldots, S_{n+1}\}$$

so that the latter set of closures (without S_k) is consistent.

Now consider the possible configurations of S_k. As it is an interior node, either its parameter space is disjoint from that of all other closures (and it stands alone) or it is dominated by some other closure S_m (with which it may be mutually dominant). If its parameter space is disjoint from all others, it is trivially consistent with the rest and we are done. If there is overlap, then choose some other closure S_m that dominates S_k (Figure 3). As S_m is a member of the consistent set of closures except for S_k, setting S_k's configuration from S_m results in a consistent set of configurations overall. We know we can do this, and that it unambiguously instantiates all parameters, because $S_m \succ S_k$ and $\mathcal{V}(S_m) \supseteq \mathcal{V}(S_k)$. □

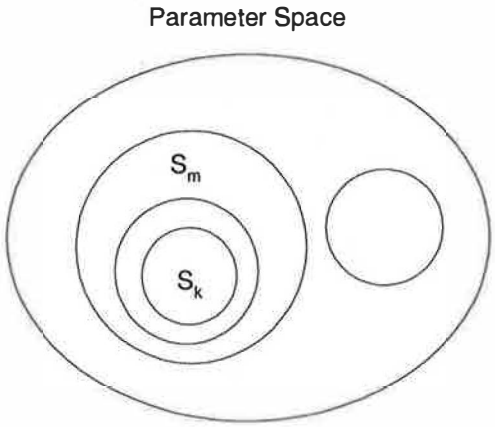


Figure 3: S_k is dominated by S_m.

The proposition describes a convenient way to assure consistency of closures from a condition on all pairs of closures:

Principle 17: To maintain consistency of closures, it is sufficient to maintain copies of parameters and values of dominated closures inside the parameter space of dominating closures.

The proposition and principle are not *optimal*. One can organize one's closures as a tree, with one closure containing the parameters for all the others, and consistency is assured. This does not rule out other schemes for assuring consistency. There are many consistent sets of closures that do not obey this stricture. There is room for much future work on the optimal structure of consistent sets of closures.

For now, without further theory, higher-level closures must maintain copies of lower-level configuration data, or face potential inconsistencies. As a side-effect, this inclusion makes dominance transitive. This copying may seem a bad thing but alas, not all hope is lost, because what dominating closures have to copy is actually rather limited. The structure imposed by the proposition leads to a strict hierarchy (tree) of closures, where one overarching closure contains (or has knowledge of) all of parameter space.

Exterior Parameters

An *exterior parameter* of a closure is a parameter that – according to the previous principle – we must copy upward in a chain of closures.

Definition 22: A set of parameters $E \subseteq \mathcal{V}_A$ is *exterior* if for all $c \in \mathcal{D}_A$, the values in

$$c|_E = \{(p, c(p)) \mid p \in E\}$$

uniquely determine the outcome of tests in $\mathcal{T}_A$.

This is a global constraint on all achievable configurations. $c|_E$ is read "c restricted to domain E." In other words, given the values of parameters in E, all policies compliant with these parameter values have identical behavior. More formally, given $E \subseteq \mathcal{V}_A$ and a configuration c, the configurations that agree with values in $c|_E$ are interchangeable. This means that

$$\{D \in \mathcal{D}_A \mid \forall p \in E, c|_E(p) = D(p)\}$$

all have identical behavior.

A set of parameters is exterior if, given these, the behavior of the closure is completely determined, regardless of the other parameters, modulo the behaviors we can observe. Interior parameter values are dependent upon the exterior parameter values. If a tree falls in the forest and no one hears it, it *does not* make a sound!

Interior parameters most frequently appear in very complex systems, e.g., web servers. For example, the port number of a virtual server is exterior; the name of its root directory may not be. The former is necessary to pass any behavioral test; the latter may change without affecting external behavior at all.

It immediately follows that:

Proposition 4: In a consistent set of closures, behavior is completely determined by the values for all exterior parameters of all closures.

Proof: Let $S_1, \ldots, S_n$ be a consistent set of closures. Let $\mathcal{V} = \bigcup_i \mathcal{V}_{S_i}$ represent the combined set of all parameters. Let $c: \mathcal{V} \rightarrow \Sigma^*$ be a consistent configuration for all parameters of all closures.

By definition of exterior parameters, in a particular closure S_i, the values of c restricted to exterior parameters $\mathcal{E}_{S_i}$ completely determine the behavior of that closure. As S_i was arbitrary, this is true overall. □

For consistency's sake, when we make copies of parameters in dominating closures, only the exterior parameters need to be copied in this manner. In other words:

Principle 18: At any level of a dominance hierarchy, all the exterior parameters of dominated closures need to be present as part of the configuration.

This does not mean that these parameters are necessarily exterior at the *current level*; they may be interior and their values under control of the dominant closure or some other dominated closure.

For example, in an intranet closure, the port on which an internal webserver runs is interior to the intranet but exterior to the servers serving the intranet.

Note that the fact that a parameter is exterior means that it determines interior parameters, not that interior parameters do not exist. The exterior command "be a fast webserver" instantiates a large number of interior parameters, including speed of CPU, disk, ethernet, etc. The latter comprise the options that one can take in creating a fast web server, and the possibility of multiple policies that accomplish this allows that "there's more than one way to do it."

Interface

So far, we have considered the internal structure of a closure, including local parameters and propagation of exterior parameters. All configuration management tools so far define their interface in terms of this parameter space. This is a complex and trying task; it is much easier to base the interface on behavior rather than parameterization.

Note that in our model, the reasonable policies $\tilde{\mathcal{P}}_A$ of a closure A map to a set of behaviors

$$\tilde{\tau}_A(\tilde{\mathcal{P}}_A) = \{ \tilde{\tau}(\mathcal{Q}) \mid \mathcal{Q} \in \tilde{\mathcal{P}}_A \}$$

ideally in a one-one fashion, because $\tilde{\mathcal{P}}_A$ is constructed as the inverse image of the test space. This means that *the behaviors determine the reasonable parameter sets that assure each behavior.* Thus it is possible to

Principle 19: Specify behavior, and allow the configuration management system to derive appropriate parameters to assure that behavior.

This idea is borrowed from Anderson [2]. In the current version of LCFG, one maintains grids of computers by specifying constraint spaces for their configuration, from which a reasonable configuration is chosen. For example, one simply asserts that "every network has a DHCP server," and the SmartFrog framework allows stations to vote on the best location for the DHCP server. In the case of LCFG, constraints are still formulated upon configuration parameters, whereas in our case, constraints ideally act upon behavior, independent of the

structure of parameter space. This is admittedly a difficult distinction to make, and one may argue that the structure of LCFG's parameter space accomplishes the same objective.

This has a subtle effect upon parameter propagation upward across dominance boundaries. We already know that it is only necessary to expose exterior parameters, and that behavior induces the appropriate parameter configurations. An upper closure that is informed enough to know the map from behavior to parameter values need not store the values; the desired behaviors are enough to store.

Comparison With Other Approaches

To understand the differences between this approach and typical configuration management systems, consider how we would configure typical network services using the new system. We decompose the problem of providing service into a dominance hierarchy in which the leaves are the files and processes that control service provision. In a typical UNIX system, these files might include /etc/inetd.conf, /etc/services, /etc/protocols, /etc/hosts.allow, /etc/hosts.deny, and other files intended to configure specific services.

In traditional configuration management, the contents of each of these files is generated by custom scripts from an overall specification [1, 2, 18, 19, 22, 23, 36] or edited in place by incremental scripts [20, 37, 38] or convergent rules [4, 5, 6, 12]. In our system, each is instead wrapped by an agent that treats the underlying file more like a database rather than a flat file. This agent takes responsibility for all changes to the file and normalizes the form of the file (by sorting) so that the effects of independent changes to the file do not depend upon the order of changes. These agents are convergent in a stronger sense than Cfengine file editing (as proven mathematically in [16]) and replace convergent and scripted file editing with a more stable and predictable alternative.

Higher-level configuration management is accomplished by agents that become clients of file-level agents. A "service" agent interacts with the file agents for the files that control a service, and with "process" agents that manages the existence and restart of particular daemons. The service agent receives data on which services should be present and propagates that to the configuration files by conversing with appropriate file agents. Non-behavioral attributes of configuration (e.g., exactly where specific files live) are handled by the closures that manage content of the files, and are no longer part of the configuration of a system.

We are all "washed by the very same rain" [29] and many recent innovations in configuration management have very similar characteristics to these agents. The behavior of agents is similar to distributed file generation procedures in LCFG [1, 2], Arusha [18, 19], and Psgconf [36], and "embody expert knowledge about how to configure a system" [39]. LCFG author Paul Anderson is considering reforming LCFG's language around specifying "constraints" rather than "parameters." In many ways, everyone is converging upon one way of thinking: specify external behavior, and leave everything else to the configuration tool.

There is one extreme contrast between our model and every other model of configuration management. In every other model, *closures are systematically violated* by reverse-engineering appropriate parameter values and setting these values directly by file generation or editing. This seems convenient but has a deadly result. It means that the administrator configuring the management tool has to have *complete knowledge* of the semantics of interior and exterior parameter space of *all* closures in order to automate administration, including interior parameter semantics for all possible versions of that parameter space (e.g., for multiple operating systems and hardware platforms). We suggest a controversial departure from this:

Principle 20: Even advanced configuration management tools should utilize conduits rather than direct file editing to manage closures.

Tools must avoid end runs around protections and integrity constraints built into closures. The management tool need not know the fine-grained semantics of interior parameters, and can concentrate on exterior behavior. The result is to naturally *distribute knowledge* about the system in two places: a high-level description of behavior in the configuration management tool, and a low-level description of how to assure behavior (via interior parameters) in the closure itself. This distribution makes it possible to configure networks with a portable high-level language that becomes customized when applied to specific architectures or platforms.

There are many examples where use of an advanced configuration management tool such as Cfengine destroys a closure. The most obvious problems occur in trying to use Cfengine for package management [20]. The master RPM repository for a particular version of RedHat Linux, together with the rpm command that unravels and installs it, is a strong closure [17], but only if the rpm command is used as intended, and only when no other effects are interposed (such as editing the files that rpm edits during post-install scripts). Interspersing arbitrary configuration commands between a series of package installs can lead to validation drift [17] and eventual failure of the system.

Initial Prototypes

To prove the concept of a closure, we decided to create three prototypes in the context of a group Masters project in System Administration at Tufts University. Concurrently, Prof. Couch continued to develop the mathematical theory of closures. The most important lesson we learned is that in a race between developing theory and prototypes, theory almost always

matures faster. The emerging theory led us to critique the initial designs of the closures in quite unexpected ways, and led to a stronger understanding of the theory as well.

We began our study of closures by trying to replace the somewhat problematic "file editing" features of Cfengine with a more strongly convergent substitute agent, through which all editing transactions could flow under control of Cfengine. This agent understands the structure of the file being edited, as well as the desired output format for the file. Upon receiving editing commands inspired by SQL, the agent parses the file, makes changes, and rewrites the file according to specifications.

The second prototype is a conduit for editing configuration files that understands policy and integrity constraints on file contents. XML [3, 9, 27, 44] declarations describe the format of the incoming file, the desired output, and the range of allowable changes to the file. The file is parsed through a stream-parser based upon Babble [13] and the output generated through use of XSLT stylesheets. This is similar to TemplateTree II [34] except that the whole configuration file is parsed and all configurable variants exposed.

The third prototype uses the first prototype to handle high-level configuration data. It expresses changes in service as changes in configuration, and converses with the relevant file and process agents to affect the changes. Its own high-level file format is amenable to editing via the second prototype, so that the circle is complete and one can specify low-level effects with high-level commands.

The focus of all these early prototypes is basic service provision and security. The long-term goal of this work is to create agents that completely encapsulate the process of configuring an Apache web server, so that virtual services can be created flawlessly without any chance of disrupting other virtual services.

Encapsulating File Editing

It is generally agreed by practitioners that on-the-fly file editing is a weakness of many configuration management strategies. Many avoid explicit editing by generating all file contents from databases. As an exercise, we studied instead how one could encapsulate file editing into a closure that protects against typical file-editing mistakes.

Atomicity

As a first attempt to solve this problem, we built a prototype *flat file configurator* (FFC) that encapsulates the idea of editing a field-based configuration file from /etc, such as services, inetd.conf, etc. This particular closure made all updates atomic. The closure is an agent that maintains the contents of a file via database-like commands with a syntax inspired by SQL. These commands include:

- load /etc/services: Make contents available for editing.
- in /etc/services: Set a context in which edits will occur.
- insert what (service='tftp', port='8888', proto='tcp') : Add a line to the in-memory version of /etc/services.
- delete where (service='dns' and proto='udp') : Delete a line from the in-memory version of /etc/services.
- update who (first 5) where (proto='tcp') what (proto='udp') : Change the first five entries in the file having protocol tcp to protocol udp.
- update who (last 1) where (service='dns') what (proto='udp'): Change the last line where service is dns to have protocol udp.
- display: Show contents of file on terminal.
- display as XML: Show contents as XML.
- write: Update original file with in-memory data.
- write to /etc/services.new: Posts results to a new file.
- end in: Terminate editing context.

These commands preserve the order of the input file when updating and deleting lines, and insert new lines at the end. Updates leave data in the same position in the file. Although the closure does not allow completely duplicate lines, it is possible to create files that have two port entries for the same service and protocol. This usually results in an invalid services file.

This is a fine first attempt, but new theory shows that it is not a particularly strong closure. There is a much better approach.

Invariance

The closure above solves the problem of editing but leaves the problem of file integrity unaddressed. A suitable closure for /etc/services must preserve three invariants of the file:

1. The file is sorted in a deterministic order by port number, protocol, then service name.
2. The file has at most one entry for each service and protocol combination.
3. There is no other content to the file.

One reason for this approach is to assure that a list of edits, applied to two copies of the same initial file, will both have identical Tripwire [43] or Aide [31] signatures.

To make a better closure, we must limit the set of commands to those that cannot violate these invariants, and construct them so that they do not:

- assert what (service='tftp', port='8888', proto='tcp'): Insures that there is a line for service tftp. This will change an existing line if it is present.
- retract where (service='tftp'): Insures that there is no line describing service tftp. This will delete a line only if present.
- commit: Commits changes to the file to disk.

These commands change content in a stateless manner [16] so that the result of editing is invariant of the order of individual assertions and retractions of data.

The lesson learned from this example is that the strength of a closure is proportional to the lack of irrelevancy in the command set and its inability to express less-than-ideal states. Expressive power is an enemy of closure and predictability. This new command set has the following properties:

- Commands always succeed and there is no error state (barring hardware failure)
- Every command preserves the invariants of keeping the file in sorted order and having one line per service.
- The commands are not based upon a parameter model of the file, but upon a *holistic* model of the meaningful states of the file.

While it has been proposed that we should think of individual files as databases [22, 23], this shows that thinking of them in that way is too powerful, and only by limiting the operations on a file to a small set that preserves invariants can one truly construct a closure on the file. The database model is too powerful and flexible to create the appropriate effect.

The above attempt is a closure, but is still not a particularly strong one. It preserves the integrity of the structure of the file, but does not reflect content policy above and beyond that integrity. To go further, knowledge of this policy and of best practices for content of /etc/services must be included in the agent. It does not matter whether the closure *understands* whether a state is valid or not; we simply have to arrange by some means that no invalid state can be assured by the agent. This is not understanding; just syntax.

Policies

The prototype above does not go far enough to eliminate errors. The strength of a true closure for /etc/services lies in its ability to avoid invalid or undesirable states. For most sites, this means that the ports of several well-known services should not be possible to change. As well, it should not be possible to delete or modify certain well-known service records whose absence will create havoc, e.g., time. These assumptions – part of typical site policies – are not representable easily in database form, and the database analogy for closures again breaks down.

To implement this kind of closure, the command set has to be insensitive to certain kinds of changes and may not require full data in order to make an assertion of state. For example, the well-behaved closure agent might presume that

```
assert service=ssh
```

always implies protocol tcp and port 22, so these do not need to be specified. If one tells this closure to

```
assert service=ssh, port=33
```

then it has every right to ignore the request and/or raise an exception. This is an assertion of invalid state according to a site policy that is itself an expression of accepted practice according to the site's practice manual. Likewise, the same closure will reject suggestions like

```
retract service=ldap
```

so that a user has some hope of logging into the host. The feature of limiting configuration actions based upon site policy was first developed in Slink [10].

Implementation

Implementing such a closure is actually relatively straightforward. One first describes the structure of a flat file as an XML declaration, as in Babble [13] (Figure 4). This defines the variant parts of a typical line in /etc/services in the same way that Babble defines the variants in a terminal transaction stream. This declaration is used both to parse and regenerate the file during each editing transaction. Similar declarations could be used to constrain content in creating the proposed changes above, including listings of defaults.

```
<file fileName="services">
  <line>
    <comment marker='#' />
    <var name='service'
         regexp="[A-Za-z0-9_-]+" />
    <whitespace/>
    <var name='port'
         regexp='[0-9]+' />
    <literal>/</literal>
    <var name='proto'/>
    <whitespace />
    <var name='alias'
         regexp='[^#]*'
         optional="true" />
    <var name='comment'
         regexp='#.*'
         optional="true" />
  </line>
</file>
```

Figure 4: Declaring database structure of the services file.

Prototype Status

This is the most evolved of all the prototypes, and is fully functional. It is currently implemented as a single Perl script with a command-line interface. A second configuration file can bind particular structural configurations to the various files in /etc (though at present, only the /etc/services interface has been extensively tested).

Critique

File-editing is a necessary process, but a poor level at which to form a really strong useful closure. Some part of a closure must understand more than the syntax of the file. Basing one's whole strategy upon editing, as in Cfengine, does not form strong closures, even if the editing itself is strongly closed as above. More constraints are required than pure syntax can provide.

Constrained Editing

The second prototype implements an interactive web-based editor for configuration files that understands structural limits of each kind of file. The editor makes it possible to add any legal entry to the file

according to a predeclared policy, but prevents adding any illegal entries to the file.

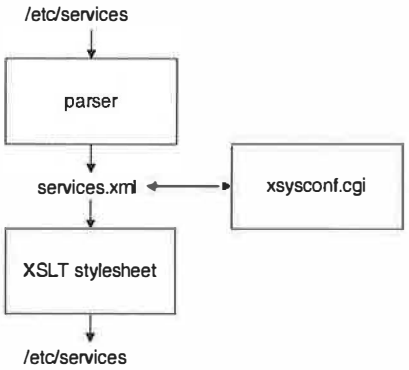


Figure 5: Editing a flat file via XML.

The parser is configured to edit a particular kind of file by specifying the constraints upon the file in XML, using a syntax again similar to that of BABBLE [13] but more complex than that for the flat-file configurator above (FFC). For /etc/services, the constraint file is shown in Figure 6. The var fields describe variant text, while the repeat, optional, choice, and text elements have the obvious meanings. The comments in each kind of variable are a description provided to the user as help text. While this file may seem complex, it is nothing more than a detailed and highly documented regular expression! It says that a services file consists of services entries, one per line, where each line consists of a service name, a port, a forward slash, a protocol name, and optionally a list of aliases and/or a comment.

Using this declaration, the parser converts the flat format of the source file to XML. The /etc/services source file:

```
ssh 22/tcp ssh-2 ssh2 # secure shell
telnet 23/tcp ncsa-telnet
```

is translated into the XML in Figure 7. It is upon this intermediate form that editing takes place. There is presently one design flaw in the prototype: if no alias exists when the initial record is created, none can be added.

The editing process is a web-based CGI script that presents editing options as pulldown menus and text fields (Figure 8). One can change or add fields by pressing the appropriate buttons or changing values and pressing "write."

Prototype Status

Currently, the only fully implemented part is the CGI-based editor for XML. The translators to and from XML are not yet implemented, though the translator into XML is almost identical with that for FFC

```
<xmft:file path="/etc/services">
  <xmft:repeat sorted-by="port" keys="service:port+prot" name="lines">
    <xmft:line>
      <xmft:var type="string" desc="service name" name="service">
        This is the service name. This should correspond to the same
        service in inetd.conf.
      </xmft:var>
      <xmft:whitespace/>
      <xmft:var type="integer" desc="ip port number" name="port">
        This is the Internet Port number of a service. The pair of
        port number and protocol must be unique in the file.
      </xmft:var>
      <xmft:text>/</xmft:text>
      <xmft:choice type="protocol name" name="prot">
        <xmft:option><xmft:text>tcp</xmft:text></xmft:option>
        <xmft:option><xmft:text>udp</xmft:text></xmft:option>
      </xmft:choice>
      <xmft:repeat>
        <xmft:whitespace/>
        <xmft:var type="string" desc="protocol alias" name="alias">
        </xmft:var>
      </xmft:repeat>
      <xmft:optional><xmft:whitespace/></xmft:optional>
      <xmft:optional>
        <xmft:text>#</xmft:text>
        <xmft:optional><xmft:whitespace/></xmft:optional>
        <xmft:var type="string" name="comment">
          This is a comment describing the service defined by this line
        </xmft:var>
      </xmft:optional>
    </xmft:line>
  </xmft:repeat>
</xmft:file>
```

Figure 6: Declaring variant structure of the services file.

above, and the XSLT for generating output is almost identical to that of Finke [23]. Thus we had high confidence in our future ability to implement these, and left them undone in this proof-of-concept prototype.

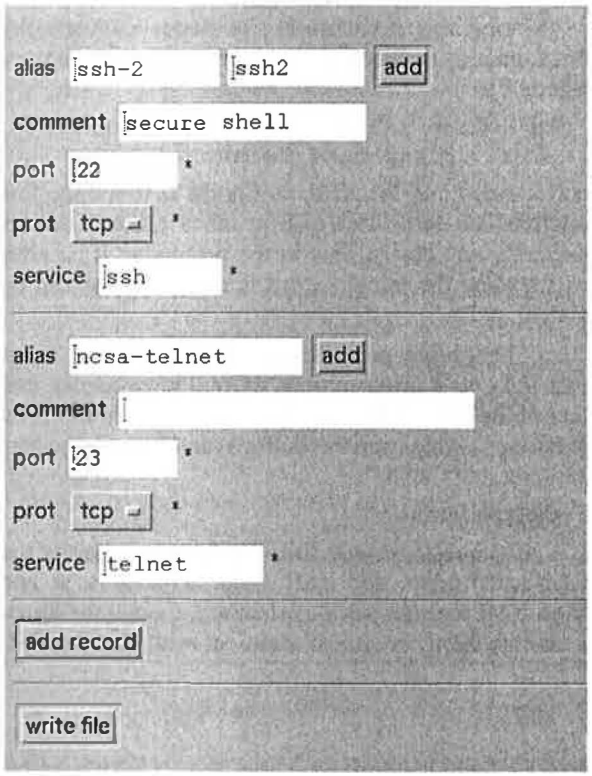


Figure 8: HTML editing form for services file.

Critique

Again, theory developed in concert with this prototype changed our thinking about it. The missing and crucial element of this editor is the ability to define constraints beyond simple syntactic correctness; the exact same constraints needed for a fully useful version of the FFC above.

Service Configuration

As an example of a high-level agent, we also implemented a service configuration agent. This was our first attempt to build a configuration environment based upon multiple agents acting together. The top-level agent is told what services to configure, and tells lower-level agents what to add to particular configuration files. While this prototype is the most challenging and least complete of the ones we attempted, it does serve to demonstrate that high-level integration is possible.

Implementation

The editor is the most complex of the prototypes, and contains three parts (Figure 5):

1. A *parser* that parses a flat file and converts it to XML.
2. A *CGI-based editor* that uses XSLT to define an interactive web-based editing session for the XML file's contents, in HTML.
3. An *XSLT renderer* that translates the XML file back to a flat file.

Architecture

The service configuration closure involves many component closures, as illustrated in the dominance diagram in Figure 9. Closures above dominate closures below.

1. The *configuration* closure dominates all others.
2. The *service* closure is the highest-level closure for services. It manages all aspects of services, including whether they are running or not, as well as current runtime state.

```
<file path="/etc/services">
  <repeat name="lines">
    <record>
      <repeat>
        <alias mode="optional" thing="var" type="string">ssh-2</alias>
        <alias mode="optional" thing="var" type="string">ssh2</alias>
      </repeat>
      <comment mode="optional" thing="var" type="string">secure shell</comment>
      <port mode="required" thing="var" type="integer">22</port>
      <prot options="tcp,udp" mode="required" thing="choice">tcp</prot>
      <service mode="required" thing="var" type="string">ssh</service>
    </record>
    <record>
      <repeat>
        <alias mode="optional" thing="var" type="string">ncsa-telnet</alias>
      </repeat>
      <comment mode="optional" thing="var" type="string"></comment>
      <port mode="required" thing="var" type="integer">23</port>
      <prot options="tcp,udp" mode="required" thing="choice">tcp</prot>
      <service mode="required" thing="var" type="string">telnet</service>
    </record>
  </repeat>
</file>
```

Figure 7: Translation of a small services file into XML.

3. The *process* closure starts and stops individual processes.

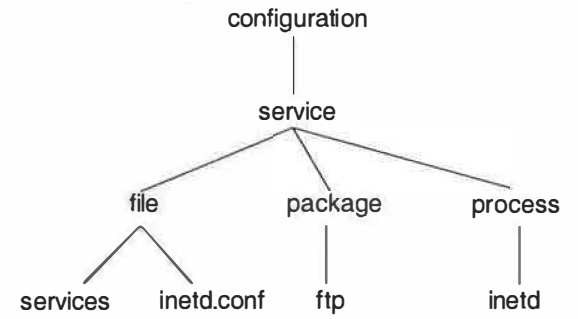


Figure 9: Dominance hierarchy for the service configuration agent.

4. The *file* closure is the lowest-level closure of the system, that only receives commands. It is responsible for editing files to contain appropriate configuration lines.
5. The *package* closure manages installation and removal of packages using RPM.

Many others are also present and planned but omitted due to space constraints. In typical use, the service closure contacts the file closure to affect changes in configuration, the process closure to start and stop service processes, and the package closure to install new service files.

All closures have the same commands assert and retract. Assert makes something available to run within the configuration, while retract makes it unavailable. All closures also have a show command that reports on current status. Most closures have start and stop, that control whether something is executing or dormant.

The Configuration Closure

The configuration closure is the main user interface to the system. For some idea of the form of the

```
assert service tftp
```

creates the service tftp on its normal port, while

```
assert service tftp port=6699
```

creates the service on a different port. The command

```
retract service tftp
```

removes a service, while

```
restart service tftp
```

administrator's interface, the command restarts all tftp daemons for an installed service tftp So far, this looks very similar to print managers such as *lpc*. The difference is in how this is implemented. Rather than doing the work in one large program, the service closure farms out work to subclosures that it dominates. For example, assert ftp results in the chain of assertions shown in Figure 10. Each line ending in a colon determines a closure context in which the commands below it are executed. Ellipses signify detail that is omitted for clarity.

The point of this process – and what distinguishes it from command-line managers such as lpc – is its structure, not its interface. High-level commands are broken down into low-level commands that include

```
configuration:
  assert service ftp
    service:
      assert ftp
        file:
          assert file "
            load /etc/services
            in /etc/services:
              insert what (service='ftp-data', ...)
              insert what (service='ftp', ...)
              write
            end in
          "
        package:
          assert ftp
        file:
          assert file "
            load /etc/inetd.conf
            in /etc/inetd.conf:
              insert what (service='ftp', ...)
              write
            end in
          "
configuration:
  start service ftp
    service:
      process start ftp
        process:
          start ftp
```

Figure 10: Communications between agents in assuring services.

interior parameters implied by the high-level commands. The echoing of the same command-line structure at each level – assert and retract – makes it easy to implement the various closures.

Critique

The only problem with this prototype is that it was the most ambitious of the group, and its design changed much more than the others during development, so that it is not yet fully functional. File editing is problematic, because the closure was built before the file editing closure was completed, and does not communicate well with the existing file editing closure. The theoretical result that exterior parameters of a closure must be present in each closure that dominates it is echoed in the structure of this prototype, even though the prototype was developed completely independent from the theory.

Conclusions

This is the beginning of a new and long journey, but the conclusion of another. For years, we struggled to define causality in a complex system [15, 17, 40]. We conclude that causality in a realistic system is a myth unless we synthesize it ourselves through the discipline of closure. Many principles of practice guide us, but there is still much work to be done, both on examples of closures and more clever principles of practice.

Future Work

It is summer, our masters project activity is a pleasant memory, and all the students involved have graduated, but obviously there is still much to be done. In the immediate future a new team of students will be completing a second generation of prototypes that are deployable in practice, by adding policy and practice awareness. On the theory side, we plan to more extensively study the algebraic properties of closure by applying the theory of semigroups [24, 28, 32], in a manner similar to that of [16].

Another knotty problem is to be able to efficiently define (and ultimately, automatically derive) the sometimes subtle maps between configuration and behavior. This is very much dependent upon the various "Book of Knowledge" projects [26, 30] to provide us with the words and practices that the closures should employ. This is really a problem in knowledge representation, and we envision using inductive logic programming [12] to complete exterior specifications with adequate values for interior parameters.

The reason that we carefully say *adequate* rather than *optimal* is that – due to its relation with graph theory – the problem of finding optimal parameter settings is often intractable. As an administrator, one would not even try; one would settle for settings that are *good enough*. Our automated agents should do likewise.

The map between exterior and interior parameters must take a concise, readable form that is easy to validate and certify. While working on our prototypes, we realized that the Stem [25] framework for network programming provides an ideal environment for exploring closures. Future development will focus on placing the closures in a Stem framework, using built-in Stem message passing to implement conduits.

It will be difficult to integrate this method of management with existing configuration management practices. The most difficult problem is to arrange isolation of closures and avoid *any* chance of corrupting a closure outside a conduit. This is beyond the capabilities of current file protection schemes, and might even require a new model of protection for filesystems similar to the model that protects memory at runtime.

We have shown so far that traditional hierarchical control flow can exhibit closure in an expedient fashion, but have not ruled out many other forms of control. The mathematical framework we developed allows many other kinds of communication, including peer-peer and even *clique*, in which one agent of a group arbitrarily appoints itself as the master of the group similarly to the way this is done in LCFG.

These are all difficult problems but – from our experience – all are tractable. Further, once each problem is solved once, the effects last forever. We think it will be worth the effort.

Some Lessons Learned

The theory of closures has some surprising ramifications for practice. Closures are only as strong as the discipline with which they are managed. This discipline includes only making changes via approved mechanisms. For example, the master RedHat repository is a closure if managed properly with rpm, but stops being a closure if one works outside rpm's framework, e.g., with make.

The most astonishing lesson learned is that common automation practices can destroy or weaken closures that already exist. For example, suppose that one wishes to install an RPM package via Cfengine. A common practice is to perform the RPM installation on one host, reverse-engineer its effects, recode the installation into a series of convergent Cfengine declarations, and use Cfengine to assure compliance with the declarations instead of running the original installation on each new host. The problem with this practice is that the original closure was only valid if RPM installation was used; this new method cannot, e.g., predict all of the potential effects of a post-install script when the script is executed on a previously unforeseen kind of host. Worse, the re-engineering has to be redone each time a new version of the package appears. This observation gives some credence to the order-based infrastructure-building strategies discussed in [37, 38].

The fact that "order matters" [38] is, however, a symptom of a deeper problem. Every configuration action is (as claimed in [38]) a computer program. As with any computer program, there is a set of preconditions that specify what must be true within a system

before the action will work. In [37, 38], one avoids facing the preconditions by replacing them with order constraint, where the appropriate order of operations is something observed in practice. This is only necessary if preconditions cannot be understood. Order only matters if we do not understand enough about preconditions to be able to vary the order. Our initial theory work suggests that preconditions can be modeled in very much the same way as behaviors; they are a tractable and fairly simple set of conditions upon a much larger and more complex system. With a full understanding of preconditions, order no longer matters. This fact merits further investigation.

Coming to Closure

The current practice of configuration management has high inertia. Many people, trained in the old ways, will find it difficult to adapt to the constraints that closures impose. Many configuration management tools will impede rather than help the process, because of their inherent inability to talk with conduits, as well as the tools' implicit support and encouragement of the closure-weakening reverse-engineering process described above. Vendors will impede the process by failing to provide closures themselves.

How can we evolve current practice toward effective closures? There are several steps:

1. Understand the concept of a closure
2. Identify and treasure the ones already present.
3. Avoid compromising existing closures.
4. Proactively look for domains of application.
5. Deploy win-win closures (that do not enforce limits on users) first.
6. Justify one-sided closures (that limit users) in terms of reduced lifecycle cost.

Our profession has not sufficiently evolved from arguing from "cost of equipment" to "cost of ownership." [35] If one's thinking is based upon cost of equipment and software, closure makes no sense. When one factors in the cost of administration, closure becomes a cost-saving mechanism.

A Vision of the Future

The evolving theory of closures promises a completely different of role for future system administrators than what they have today. Networks will not consist of "machines," but rather will be built of "closures" that span machines. When one places a new machine into a network, the closures operating within the network will discover it and configure it to talk with them. If this succeeds, no administrator intervention will be required; the machine will simply start working properly. If it fails, one must debug the closures (or the machine) to see why the closures did not properly integrate the machine into the network and install themselves for future use.

In the new scheme, the administrator's job is no longer to remember complex and detailed minutiae, but instead to keep an internal model of the big picture and how things fit together in a large and universal block diagram. Low-level knowledge (how a closure works) is much less important than high-level knowledge (which closures are compatible with which others). Dependency analysis and troubleshooting become jobs of specialists rather than generalists, and are not part of typical practice. Typical practice is to choose closures that work, install them, and watch them work. If they do not work properly, they must be incompatible, and one then chooses another more compatible set. Troubleshooting is to some extent replaced by shopping. Patching and upgrades are largely software functions rather than job functions.

Another fundamental difference in this brave new world is that one must systematically avoid working around the protections created by a closure. To publish content on the world-wide web, one tells a closure to publish it, and never tries this process by hand. There is only one defense against users who would work around closures, which is to make this impossible by, e.g., placing web servers on machines to which users do not have shell access.

The most profound change is in how the administrator can view the profession. We have always thought that system administration was in actuality community building: to paraphrase Burgess, "forming cooperative communities of people and machines." [7] With closures, this community building becomes the substance of the job as well as the dream. Closures are community members that take some of the worry out of keeping the community working properly.

With closures, we can begin to evolve to a methodology for system administration that is more predictable. This will not make the job of administrator obsolete; just a lot more pleasant and rewarding.

Acknowledgements

This work is not the product of individuals, but of a coordinated inquiry among a large and diverse intellectual community. We are grateful to the Large-Scale System Configuration mailing list (lssconf) for providing sustained discussion of the strategies for configuration management and their impact. Yizhan Sun provided detailed, in-depth comments and corrections to the mathematics. Special thanks to Rob Kolstad and Adam Moskowitz for insightful comments (on the first sketch of the paper) that pushed our thinking to a new level. Special thanks to George Leger and shepherd Æleen Frisch for extremely careful proofreading and copy editing. Special thanks to Mark Burgess, Paul Anderson, Steve Traugott, Mark Roth, and Luke Kanies, for providing the main fuel for this debate with their ground-breaking work. This work was supported in part by a generous equipment grant from Network Appliance Corporation.

Author Biographies

Alva L. Couch attended the North Carolina School of the Arts in his home state as a high school

major in bassoon and contrabassoon performance. He received an S.B. in Architecture from M. I. T. in 1978, after which he worked for four years as a systems analyst and administrator at Harvard Medical School.

Returning to school, he received an M.S. in Mathematics from Tufts in 1987 and a Ph.D. in Mathematics from Tufts in 1988. He joined the the faculty of Tufts Department of Computer Science in the fall of 1988, and is currently an Associate Professor of Computer Science.

Prof. Couch is the author of several software systems for visualization and system administration, including Seecube (1987), Seeplex (1990), Slink (1996), Distr (1997), and Babble (2000). He can be reached by surface mail at the Department of Computer Science, 161 College Avenue, Tufts University, Medford, MA 02155. He can be reached via electronic mail as couch@cs.tufts.edu. His work phone is (617)627-3674.

John Hart has worked with Prof. Couch on configuration management for more than five years, first authoring the 'YoKeL' configuration language (Yet anOther Configuration Engine and Language) front-end to Prof. Couch's Prolog system administration interface. He received a Bachelor's degree in Computer Engineering from Tufts University in 2001 and a Masters degree in Computer Science from Tufts in 2003. While at Tufts, he worked as an undergraduate and graduate teaching assistant, LAN administrator, and as a consultant on web design, system administration, and other tasks. John can be reached via surface mail at 215 N. 2nd St., Olean, NY 14760, or electronically at jhart@cs.tufts.edu.

A native of Montana, Elizabeth G. Idhaw lived in Sanford, Maine until graduating high school. Her interest in computer engineering brought her to Lehigh University in Pennsylvania, where she earned a Bachelor of Science degree in that field. After graduating in 1998, she moved to Massachusetts to work as an AlphaServer Hardware Engineer for Compaq Computer Corporation in Marlborough. She returned to school in 2001 for graduate study in Computer Science at Tufts University. With a Master of Science degree under her belt, she began work as a Senior Network and Distributed Systems Engineer for The Mitre Corporation of Bedford, Massachusetts. She can be reached by electronic mail at greenlee@cs.tufts.edu.

Dominic H. Kallas hails from Arlington, Massachusetts. He received his Bachelor of Science degree in Electrical Engineering from Tufts University in 1999 and continued to a Master of Science degree in Electrical Engineering, completed in 2003. His concentration is in Wireless Communications, and he is interested in network design and signal processing. He welcomes e-mail at dkallas@cs.tufts.edu.

References

[1] Anderson, P., "Towards a High-Level Machine Configuration System," *Proc. LISA-VIII*, Usenix Assoc., 1994.

[2] Anderson, P., P. Goldsack, and J. Patterson, "SmartFrog Meets LCFG: Autonomous Reconfiguration with Central Policy Control," *Proc. LISA XVII*, USENIX Assoc., pp. 219-228, San Diego, CA, 2003.

[3] Bohlman, E., "Parsing XML, Part 1," http://www.perlmonth.com/perlmonth/issue4/perl_xml.html.

[4] Burgess, Mark, "A Site Configuration Engine," *Computing Systems*, Vol. 8, 1995.

[5] Burgess, Mark and R. Ralston, "Distributed Resource Administration Using Cfengine," *Software: Practice and Experience*, Vol. 27, 1997.

[6] Burgess, Mark, "Computer Immunology," *Proc. LISA-XII*, Boston, MA, Usenix, 1998.

[7] Burgess, Mark, "Theoretical System Administration," *Proc. LISA-XIV*, New Orleans, LA, Usenix, 2000.

[8] Cons, Lionel and Piotr Poznanski, "Pan: A High-Level Configuration Language," *Proc. LISA XVI*, USENIX, Philadelphia, PA, 2002.

[9] Cooper, C., "XML::Parser – A Perl Module for Parsing XML Documents," http://search.cpan.org/author/COOPERCL/XML-Parser-2.31/Parser.pm.

[10] Couch, Alva, "SLINK: Simple, Effective Filesystem Maintenance Abstractions for Community-based Administration," *Proc. LISA X*, Usenix, 1996.

[11] Couch, Alva, "Chaos out of Order: A Simple, Scalable File Distribution Facility for 'Intentionally Heterogeneous' Networks," *Proc. LISA XI*, Usenix, 1997.

[12] Couch, Alva and M. Gilfix, "It's Elementary, Dear Watson: Applying Logic Programming to Convergent System Management Processes," *Proc. LISA XIII*, Usenix, 1999.

[13] Couch, Alva, "An Expectant Chat about Script Maturity," *Proc. LISA XIV*, Usenix, 2000.

[14] Couch, Alva and Noah Daniels, "The Maelstrom: Network Service Debugging via 'Ineffective Procedures'," *Proc. LISA XV*, Usenix, 2001.

[15] Couch, Alva and Y. Sun, "Global Impact Analysis of Dynamic Library Dependencies," *Proc. LISA XV*, Usenix, San Diego, CA, 2001.

[16] Couch, A. and Y. Sun, "On the Algebraic Structure of Convergence," to appear in *Proc. DSOM'03*, Elsevier, Heidelberg, DE, Oct., 2003.

[17] Hart, J. and J. D'Amelia, "An Analysis of RPM Validation Drift," *Proc. LISA XVI*, Usenix Assoc., San Diego, CA, 2002.

[18] Holgate, M. and W. Partain, "The Arusha Project: A Framework for Collaborative Unix System Administration," *Proc. LISA XV*, Usenix, San Diego CA, 2001.

[19] Holgate, M., W. Partain, et al., *The Arusha Project Web Site* http://ark.sourceforge.net.

[20] Kanies, L., "Practical and Theoretical Experience with ISconf and Cfengine," *Proc. LISA XV*, USENIX Assoc., San Diego CA, 2001.

[21] Sandnes, Frode Eika, "Scheduling Partially Ordered Events in a Randomised Framework – Empirical Results and Implications for Automatic Configuration Management," *Proc. LISA XV*, Usenix, San Diego CA, 2001.

[22] Finke, Jon, "An Improved Approach for Generating Configuration Files from a Database," *Proc. LISA XIV*, Usenix, 2000.

[23] Finke, Jon, "Generating Configuration Files: The Director's Cut," *Proc. LISA XVII*, USENIX Assoc., pp. 201-210, San Diego, CA, 2003.

[24] Grillet, P. A., *Semigroups: An Introduction to the Structure Theory*, Marcel Dekker, Inc, New York, NY, 1995.

[25] Guttman, U., "Stem: A Sysadmin Enabler," *Proc. LISA XVI*, Usenix, Philadelphia, PA, 2002.

[26] Halprin, G., et al., "SA-BOK (The Systems Administration Body of Knowledge)," http://www.sysadmin.com.au/sa-bok.html.

[27] Harold, E. and S. Means, "XML in a Nutshell, 2nd Edition," O'Reilly, Inc, 2002.

[28] Howie, J. M., *An Introduction to Semigroup Theory*, Academic Press, 1976.

[29] Humphries, Pat, *Same Rain (Audio CD)*, Moving Forward Music, 1992.

[30] Kolstad, R., et al., *The Sysadmin Book of Knowledge Gateway*, (private site).

[31] Lehti, Rami, "AIDE – Advanced Intrusion Detection Environment," http://www.cs.tut.fi/rammer/aide.html.

[32] Ljapin, E. S., *Semigroups*, American Mathematical Society, Providence, RI, 1963.

[33] Logan, Mark, Matthias Felleisen, and David Blank-Edelman, "Environmental Acquisition in Network Management" *Proc. LISA XVI*, Usenix, Philadelphia, PA, 2002.

[34] Oetiker, T.,"TemplateTree II: The Post-Installation Setup Tool," *Proc. LISA XV*, Usenix, San Diego, CA, 2001.

[35] Patterson, J., "A Simple Model of the Cost of Downtime," *Proc. LISA XVI*, Usenix, Philadelphia, PA, 2002.

[36] Roth, M. D., "Preventing Wheel Reinvention: The Psgconf System Configuration Framework," *Proc. LISA XVII*, pp. 211-218, USENIX, San Diego, CA, 2003.

[37] Traugott, Steve and Joel Huddleston, "Bootstrapping an Infrastructure," *Proc LISA XII*, Usenix, Boston, MA, 1998.

[38] Traugott, Steve and Lance Brown, "Why Order Matters: Turing Equivalence in Automated Systems Administration," *Proc. LISA XVI*, Usenix, Philadelphia, PA, 2002.

[39] Sapuntzakis, C., D. Brumley, R. Chandra, N. Zeldovich, J. Chow, J. Norris, M. S. Lam, and M. Rosenblum, "Virtual Appliances for Deploying and Maintaining Software," *Proc. LISA XVII*, pp. 186-200, USENIX, San Diego, CA, 2003.

[40] Wang, Yi-Min, Chad Verbowski, John Dunagan, Yu Chen, Chun Yuan, Helen J. Wang, and Zheng Zhang, "STRIDER: A Black-box, State-based Approach to Change and Configuration Management and Support," *Proc. LISA XVII*, pp. 165-178, Usenix, San Diego, CA, 2003.

[41] Watt, D., *Programming Language Syntax and Semantics*, Prentice Hall, 1991.

[42] *The Linux Standard Base Project*, http://www.linuxbase.org.

[43] Tripwire, Inc, *The Tripwire Security Scanner*, http://www.tripwire.com.

[44] XML Working Group, *XML Schema Specification*, http://www.w3c.org.

Archipelago: A Network Security Analysis Tool

Tuva Stang, Fahimeh Pourbayat, Mark Burgess, Geoffrey Canright, Kenth Engø, and Åsmund Weltzien – Oslo University College

ABSTRACT

Archipelago is system analysis and visualization tool which implements several methods of automated resource and security analysis for human-computer networks; this includes physical networks, social networks, knowledge networks and networks of clues in a forensic analysis. Access control, intrusions and social engineering can be discussed in the framework of graphical and information theoretical relationships. Groups of users and shared objects, such as files or conversations, provide communications channels for the spread of both authorized and unauthorized information. We present a Java based analysis tool that evaluates numerical criteria for the security of such systems and implements algorithms for finding the vulnerable points.

Introduction

Network security is a subject that can be discussed from many viewpoints. Many discussions focus entirely upon the technologies that protect individual system transactions, e.g., authentication methods, ciphers and tunnels. Less attention has been given to the matter of *security management*, where a general theoretical framework has been lacking.

In this work, we explore two theoretical methods to estimate *systemic security*, as opposed to system component security. describe a tool (Archipelago) for scanning systems, calculating and visualizing the data and testing the results.

Our paper starts with the assumption that security is a property of an *entire system* [1] and that covert channels, such as social chatter and personal meetings, are often viable ways to work around so-called strong security mechanisms. File access security is a generic representation of communication flow around a system, and we use it as a way of discussing several other problems. Other issues like social engineering have previously been notoriously difficult to address in quantitative terms, but fit easily into our discussion. We have made some progress in this area by applying graph theoretical techniques to the analysis of systems [2]. In this paper we implement a tool for using these techniques and demonstrate its use in a number of examples.

The paper begins with a brief discussion of the graph-theoretical model of security, and how it is used to represent associations that lead to the possible communication of data. Next we consider how complex graphs can be easily represented in a simplified visual form. The purpose of this is to shed light on the logical structure of the graph, rather than its raw topological structure. We describe a method of eigenvector centrality for ranking nodes according to their importance, and explain how this can be used to organize the graph into a logical structure. Finally, we discuss the problem of how easily information can flow through a system and find criteria for total penetration of information.

Graphs

A graph is a set of nodes joined together by edges or arcs. Graph theoretical methods have long been used to discuss issues in computer security [3, 4], typically trust relationships and restricted information flows (privacy). To our knowledge, no one has considered graphical methods as a practical tool for performing a partially automated analysis of real computer system security. Computer systems can form relatively large graphs. The Internet is perhaps the largest graph that has ever been studied, and much research has been directed at analyzing the flow of information through it. Research shows that the Internet [5] and the Web [6] (the latter viewed as a directed graph) each have a power-law degree distribution. Such a distribution is characteristic [7, 8, 9] of a self-organized network, such as a social network, rather than a purely technological one. Increasingly we see technology being deployed in a pattern that mimics social networks, as humans bind together different technologies, such as the Internet, the telephone system and verbal communication.

Social networks have may interesting features, but a special feature is that they do not always have a well defined center, or point of origin; this makes them highly robust to failure, but also extremely transparent to attack [10]. A question of particular interest to a computer security analyst, or even a system administrator deploying resources is: can we identify likely points of attack in a general network of associations, and use this information to build analytical tools for securing human-computer systems?

Associations

Users relate themselves to one another by file sharing, peer groups, friends, message exchange, etc. Every such connection represents a potential information flow. An analysis of these can be useful in several instances:

- For finding the weakest points of a security infrastructure for preventative measures.
- In forensic analysis of breaches, to trace the impact of radiated damage at a particular point, or to trace back to the possible source.

Communication takes place over many channels, some of which are controlled and others that are *covert*. A covert channel is a pathway for information that is not subject to security controls.

Our basic model is of a number of *users*, related by associations that are mediated by human-computer *resources*. The graphs we discuss in this paper normally represent a single organization or computer system. We do not draw any nodes for outsiders; rather we shall view outsiders as a kind of reservoir of potential danger in which our organization is immersed.

In the simplest case, we can imagine that users have access to a number of files. Overlapping access to files allow information to be passed from user to user: this is a channel for information flow. For example, consider a set of F files, shared by U users (see Figure 1).

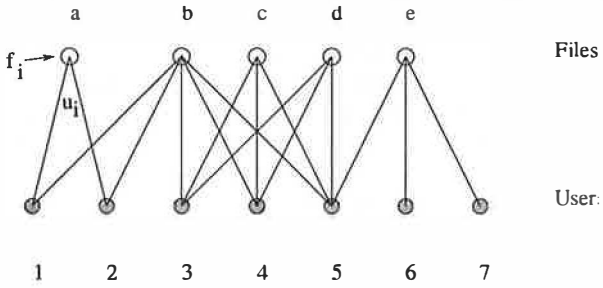


Figure 1: Users (dark spots) are associated with one another through resources (light spots) that they share access to. Each light spot contains f_i files or sub-channels and defines a group i, through its association with u_i links to users. In computer parlance, they form 'groups.'

Here we see two kinds of object (a bi-partite graph), connected by links that represent associations. A bipartite form is useful for theoretical discussions, but in a graphical tool it leads to too much 'mess' on screen. Bi-partite graphs have been examined before to provide a framework for discussing security [11]. We can eliminate the non-user nodes by simply coloring the links to distinguish them, or keeping their character solely for look-up in a database.

Any channel that binds users together is a potential covert security breach. Since we are estimating the probability of intrusion, all of these must be considered. For example, a file, or set of files, connected to several users clearly forms a *system group*, in computer parlance. In graph-theory parlance the group is simply a *complete subgraph* or *clique*. In reality, there are many levels of association between users that could act as channels for communication:

- Group work association (access).
- Friends, family or other social association.
- Physical location of users.

In a recent security incident at a University in Norway, a cracker gained complete access to systems because all hosts had a common root password. This is another common factor that binds 'users' at the host level, forming a graph that looks like a giant central hub. In a *post factum* forensic investigation, all of these possible routes of association between possible perpetrators of a crime are potentially important clues linking people together. Even in an *a priori* analysis such generalized networks might be used to address the likely targets of social engineering.

Each user naturally has a number of file objects that are private. These are represented by a single line from each user to a single object. Since all users have these, they can be taken for granted and removed from the diagram in order to emphasize the role of more special hubs (see Figure 2).

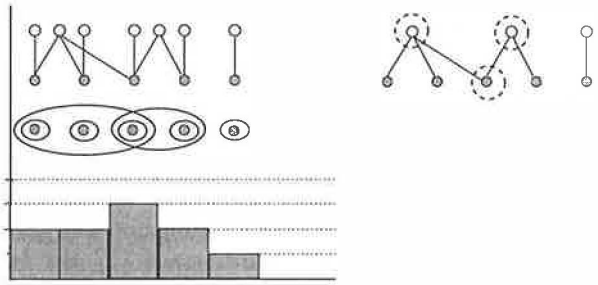


Figure 2: An example of the simplest level at which a graph may be reduced to a skeleton form and how hot-spots are identified. This is essentially a histogram, or 'height above sea-level' for the contour picture.

The resulting contour graph, formed by the Venn diagrams, is the first indication of potential hot-spots in the local graph topology. Later we can replace this with a better measure – the 'centrality' or 'well-connectedness' of each node in the graph.

Visualizing Graphs in Shorthand

The complexity of the basic bi-partite graph and the insight so easily revealed from the Venn diagrams beg the question: is there a simpler representation of the graphs that summarizes their structure and which highlights their most important information channels? An important clue is provided by the Venn diagrams; these reveal a convenient level of detail in simple cases.

Let us define a simplification procedure based on this.

Trivial group: An ellipse that encircles only a single user node is a trivial group. It contains only one user.

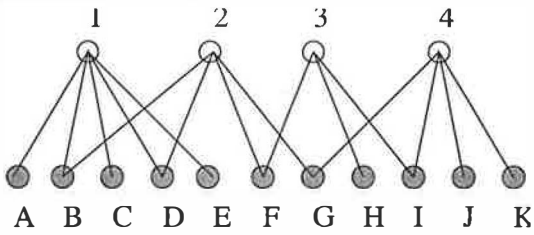


Figure 3: The example bi-partite graph from [12] serves as an example of the shorthand procedure.

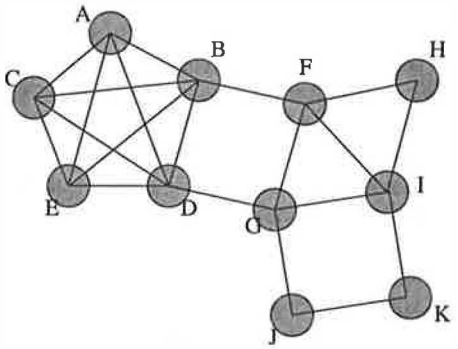


Figure 4: A 'one-mode' projection of the graph in Figure 3, as given by [12] is formed by the elimination of the intermediary nodes. Note that bi-partite cliques in the original appear here also as cliques.

Elementary group: For each file node *i*, obtain the maximal group of users connected to the node and encircle these with a suitable ellipse (as in Figure 2). An ellipse that contains only trivial groups, as subgroups, is an elementary group.

Our aim in simplifying a graph is to organize the graph using the low resolution picture generated by a simplification rule.

Simplification rule: For each file node *i*, obtain the maximal group of users connected to the node and encircle these with a suitable ellipse or other envelope (as in Figure 2). Draw a super-node for each group, labelled by the total degree of group (the number of users within it). For each overlapping ellipse, draw an unbroken line between the groups that are connected by an overlap. These are cases where one or more users belongs to more than one group, i.e., there is a direct association. For each ellipse that encapsulates more than one elementary groups, draw a dashed line.

As a further example, we can take the graph used in [12]. Figure 3 shows the graph from that reference. Figure 4 shows the same graph after eliminating the intermediate nodes. Finally, Figures 5 and 6 show this graph in our notation (respectively, the Venn diagram and the elementary-group shorthand).

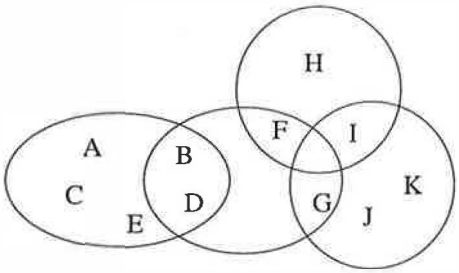


Figure 5: The Venn diagram for the graph in Figure 3 shows simple and direct associations that resemble the one-mode projection, without details.

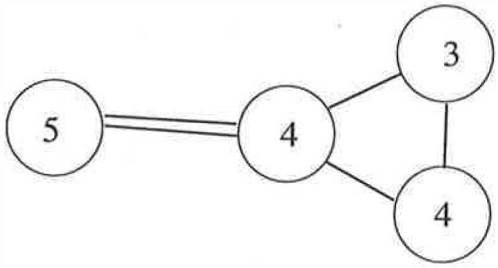


Figure 6: The final compressed form of the graph in Figure 3 eliminates all detail but retains the security pertinent facts about the graph.

The shorthand graphs (as in Figure 6) may be useful in allowing one to see more easily when a big group or a small group is likely to be infected by bad information. They also identify the logical structure of the nodes clearly. However, this procedure is complex and work intensive in any large graph. We therefore introduce a more general and powerful method that can be used to perform the same organization. This method identifies coarse logical regions in a graph by identifying nodes that are close to particularly central or important nodes and then finding those nodes that connect them together.

Node Centrality and the Spread of Information

In this section, we consider the *connected* components of networks and propose criteria for deciding which nodes are most likely to infect many other nodes, if they are compromised. We do this by examining the relative connectivity of graphs along multiple pathways.

Degree of a node: In a non-directed graph, the number of links connecting node i to all other nodes is called the degree k_i of the node.

What are the best connected nodes in a graph? These are certainly nodes that an attacker would like to identify, since they would lead to the greatest possible access, or spread of damage. Similarly, the security auditor would like to identify them and secure them, as far as possible. From the standpoint of security, then, important nodes in a network (files, users, or groups in the shorthand graph) are those that are 'well-

connected.' Therefore we seek a precise working definition of 'well-connected,' in order to use the idea as a tool for pin-pointing nodes of high security risk.

A simple starting definition of well-connected could be 'of high degree': that is, count the neighbors. We want however to embellish this simple definition in a way that looks beyond just nearest neighbors. To do this. we borrow an old idea from both common folklore and social network theory [13]: an important person is not just well endowed with connections, but is well endowed with connections to important persons.

The motivation for this definition is clear from the example in Figure 8. It is clear from this figure that a definition of 'well-connected' that is relevant to the diffusion of information (harmful or otherwise) must look beyond first neighbors. In fact, we believe that the circular definition given above (important nodes have many important neighbors) is the best starting point for research on damage diffusion on networks.

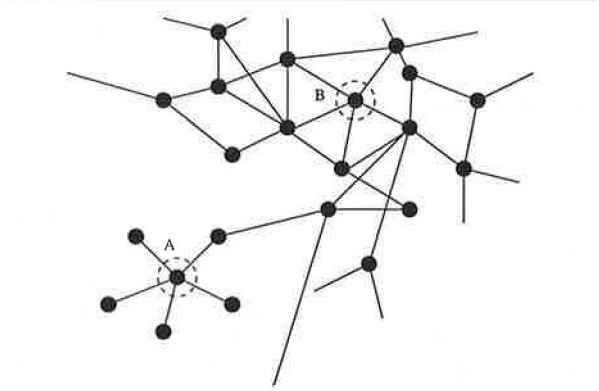


Figure 7: Nodes A and B are both connected by five links to the rest of the graph, but node B is clearly more important to security because its neighbors are also well connected.

Now we make this circular definition precise. Let v_i denote a vector for the importance ranking, or connectedness, of each node i. Then, the importance of node i is proportional to the sum of the importances of all of i's nearest neighbors:

$$v_i \propto \sum_{j = \text{neighbors of } i} v_j \quad . \tag{1}$$

This may be written as

$$v_i \propto \sum_j A_{ij} v_j \quad , \tag{2}$$

where A is the *adjacency matrix*, whose entries A_{ij} are 1 if i is a neighbor of j, and 0 otherwise. Notice that this self-consistent equation is scale invariant; we can multiply $\vec{v}$ by any constant and the equation remains the same. We can thus rewrite eqn. (2) as

$$a\vec{v} = \lambda\vec{v} \quad , \tag{3}$$

and, if non-negative solutions exist, they solve the self-consistent sum; i.e., the importance vector is hence an eigenvector of the adjacency matrix A. If A is an $N \times N$ matrix, it has N eigenvectors (one for each node in the network), and correspondingly many eigenvalues. The eigenvector of interest is the principal eigenvector, i.e., that with highest eigenvalue, since this is the only one that results from summing all of the possible pathways with a positive sign. The components of the principal eigenvector rank how 'central' a node is in the graph. Note that only ratios v_i / v_j of the components are meaningfully determined. This is because the lengths $v^i v_i$ of the eigenvectors are not determined by the eigenvector equation.

This form of well-connectedness is termed 'eigenvector centrality' [13] in the field of social network analysis, where several other definitions of centrality exist. For the remainder of the paper, we use the terms 'centrality' and 'eigenvector centrality' interchangeably.

We believe that nodes with high eigenvector centrality play a important role in the diffusion of information in a network. However, we know of few studies (see [14]) which test this idea quantitatively. We have proposed this measure of centrality as a diagnostic instrument for identifying the best connected nodes in networks of users and files [2, 15].

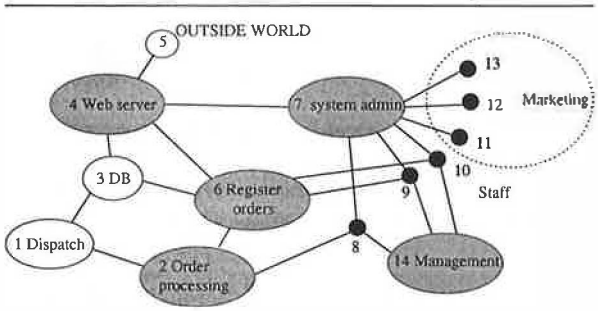


Figure 8: Unstructured graph of a human-computer system – an organization that deals with Internet orders and dispatches goods by post.

When a node has high eigenvector centrality (EVC), it *and its neighborhood* have high connectivity. Thus in an important sense EVC scores represent neighborhoods as much as individual nodes. We then want to use these scores to define clusterings of nodes, with as little arbitrariness as possible. (Note that these clusterings are not the same as user groups – although such groups are unlikely to be split up by our clustering approach.)

To do this, we define as Centers those nodes whose EVC is higher than any of their neighbors' scores (local maxima). Clearly these Centers are important in the flow of information on the network. We also associate a Region (subset of nodes) with each Center. These Regions are the clusters that we seek. We find that more than one rule may be reasonably defined to assign nodes to Regions; the results differ in detail, but not qualitatively. One simple rule is to use distance (in hops) as the criterion: a node belongs to a given Center (i.e., to its Region) if it is closest (in number of hops) to that Center. With this

rule, some nodes will belong to multiple regions, as they are equidistant from two or more Centers. This set of nodes defines the Border set.

The picture we get then is of one or several regions of the graph which are well-connected clusters – as signalled by their including a local maximum of the EVC. The Border then defines the boundaries between these regions. This procedure thus offers a way of coarse-graining a large graph. This procedure is distinct from that used to obtain the shorthand graph; the two types of coarse-graining may be used separately, or in combination.

Centrality Examples

To illustrate this idea, consider a human-computer system for Internet commerce depicted in Figure 8. This graph is a mixture of human and computer elements: departments and servers. We represent the outside world by a single outgoing or incoming link (node 5).

The organization consists of a web server connected to a sales database, that collects orders which are then passed on to the order registration department. These collect money and pass on the orders to order processing who collect the orders and send them to dispatch for postal delivery to the customers. A marketing department is linked to the web server through the system administrator, and management sits on the edge of the company, liaising with various staff members who run the departments.

Let us find the central resource sinks in this organization, first assuming that all of the arcs are equally weighted, i.e., contribute about the same amount to the average flow through the organization. We construct the adjacency matrix, compute its principal eigenvector and organize the nodes into regions, as described above. The result is shown in Figure 9.

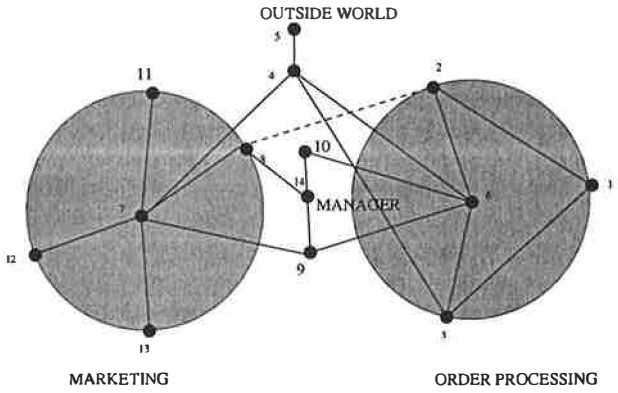


Figure 9: A centrality-organized graph showing the structure of the graph centered around two local maxima or 'most important' nodes, that are the order registration department and the system administrator. There are also four bridge nodes and a bridging link between the regions.

Node 7 is clearly the most central. This is the system administrator. This is perhaps a surprising result for an organization, but it is a common situation where many parts of an organization rely on basic support services to function, but at an unconscious level. This immediately suggests that system administration services are important to the organization and that resources should be given to this basic service. Node 6 is the next highest ranking node; this is the order registration department. Again, this is not particularly obvious from the diagram alone: it does not seem to be any more important than order processing. However, with hindsight, we can see that its importance arises because it has to liaise closely with all other departments.

Using the definitions of regions and bridges from the previous section, we can redraw the graph using centrality to organize it. The result is shown in Figure 9. The structure revealed by graph centrality accurately reflects the structure of the organization: it is composed largely of two separate enterprises: marketing and order processing. These departments are bound together by certain bridges that include management and staff that liaise with the departments. Surprisingly, system administration services fall at the center of the staff/marketing part of the organization. Again, this occurs because it is a critical dependency of this region of the system. Finally the web server is a bridge that connects both departments to the outside world – the outside hanging on at the periphery of the systems.

To illustrate the ideas further we present data from a large graph, namely, the Gnutella peer-to-peer file-sharing network, viewed in a snapshot taken November 13, 2001 [16]. In this snapshot the graph has two disconnected pieces – one with 992 nodes, and one with three nodes. Hence for all practical purposes we can ignore the small piece, and analyze the large one. Here we find that the Gnutella graph is very well-connected. There are only two Centers, hence only two natural clusters. These regions are roughly the same size (about 200 nodes each). This means, in tum, that there are many nodes (over 550!) in the Border.

In Figure 10 we present a visualization of this graph, using Centers, Regions, and the Border as a way of organizing the placement of the nodes using our Archipelago tool [17].

Both the figure and the numerical results support our description of this graph as well-connected: it has only a small number of Regions, and there are many connections (both Border nodes, and links) between the Regions. We find these qualitative conclusions to hold for other Gnutella graphs that we have examined. Our criteria for a well-connected graph are consonant with another one, namely, that the graph has a power-law node degree distribution [10]. Power-law graphs are known to be well-connected in the sense that they remain connected even after the random removal of a

significant fraction of the nodes. And in fact the (self-organized) Gnutella graph shown in Figure 10 has a power-law node degree distribution.

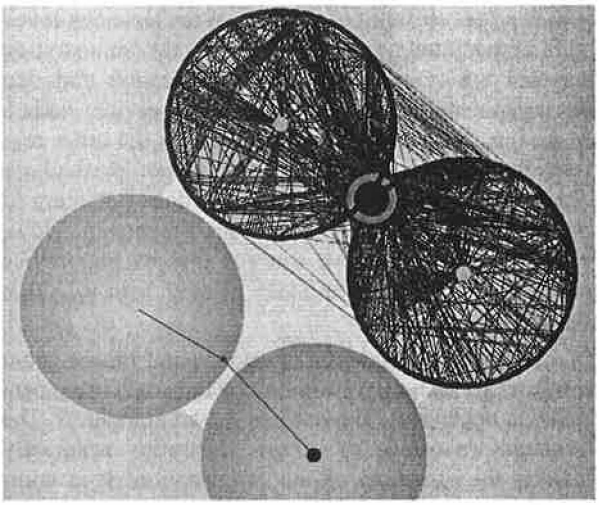


Figure 10: A top level, simplified representation of Gnutella peer to peer associations, organized around the largest centrality maxima. The graphs consists of two fragments, one with 992 nodes and one of merely three nodes and organizes the graph into Regions. The upper connected fragment shows two regions connected by a ring of bridge nodes.

We believe that poorly-connected (but still percolating) graphs will be revealed, by our clustering approach, to have relatively many Centers and hence Regions, with relatively few nodes and links connecting these Regions. Thus, we believe that the calculation of eigenvector centrality, followed by the simple clustering analysis described here (and in more detail in [15]), can give highly useful information about how well connected a graph is, which regions naturally lie together (and hence allow rapid spread of damaging information), and where are the boundaries between such easily-infected regions. All of this information should be of utility in analyzing a network from the point of view of security.

Percolation: The Spread of Information in the Graph

How many links or channels can one add to a graph, at random, before the system becomes essentially free of barriers? This question is known as the percolation problem and the breakdown of barriers is known as the formation of a *giant cluster* in the graph.

A graph is said to *percolate* if every node can reach every other by some route. This transition point is somewhat artificial for use as a management criterion, since links are constantly being made and broken, particularly in a mobile partially-connected environment of modern networks. Rather we are interested in average properties and probabilities.

One of the simplest types of graph is the hierarchal tree. Hierarchical graphs are not a good model of user-file associations, but they are representative of many organizational structures. A very regular hierarchical graph in which each node has the same degree (number of neighbors) is known as the Cayley tree. Studies of percolation phase transitions in the Cayley model can give some insight into the computer security problem: at the 'percolation threshold' essentially all nodes are connected in a 'giant cluster' – meaning that damage can spread from one node to all others. For link density (probability) below this threshold value, such widespread damage spreading cannot occur.

For small, fixed graphs there is often no problem in exploring the whole graph structure and obtaining an exact answer to this question. The most precise small-graph criterion for percolation comes from asking how many pairs of nodes, out of all possible pairs, can reach one another in a finite number of hops. We thus define the ratio R_C of connected pairs of nodes to the total number of pairs that could be connected:

$$R_C = \sum_{i=\text{clusters}} \frac{\frac{1}{2} n_i(n_i - 1)}{\frac{1}{2} N(N-1)} = 1 \quad . \tag{4}$$

This is simply the criterion that the graph be connected.

If we wish to simplify this rule for ease of calculation, we can take $n_i \approx L_i + 1$, where L_i is the number of links in cluster i. Then, if L is the total number of links in the graph, criterion (4) becomes

$$R_L = \frac{L(L+1)}{N(N-1)} > 1 \quad . \tag{5}$$

Thus we have one 'naive' small-graph test which is very simple, and one 'exact' criterion which requires a little more work to compute.

The problem with these criteria is that one does not always have access to perfect information about real organizations. Even if such information were available, security administrators are not so much interested in what appears to be an accurate snapshot of the present, as in what is likely to happen in the near future. Socially motivated networks are not usually orderly, like hierarchical trees, but have a strong random component. We therefore adapt results from the theory of random graphs to obtain a statistical estimate for the likelihood of percolation, based on remarkably little knowledge of the system.

To study a random graph, all we need is an estimate or knowledge of their degree distributions. Random graphs, with arbitrary node degree distributions p_k have been studied in [12], using the method of generating functionals. This method uses a continuum approximation, using derivatives to evaluate probabilities, and hence it is completely accurate only in the continuum limit of very large number of nodes N.

We shall not reproduce here the argument of [12] to derive the condition for the probable existence of a

giant cluster, but simply quote it for a uni-partite random graph with degree distribution p_k.

Result 1: The large-graph condition for the existence of a giant cluster (of infinite size) is simply

$$\sum_k k(k-2)p_k \geq 0 \quad . \qquad (6)$$

This provides a simple test that can be applied to a human-computer system, in order to estimate the possibility of complete failure via percolating damage. If we only determine the p_k, then we have an immediate machine-testable criterion for the possibility of a systemwide security breach.

The problem with the above expression is clearly that it is derived under the assumption of there being a smooth differentiable structure to the average properties of the graphs. For a small graph with N nodes the criterion for a giant cluster becomes inaccurate. Clusters do not grow to infinity, they can only grow to size N at the most, hence we must be more precise and use a dimensionful scale rather than infinity as a reference point. The correction is not hard to identify; the threshold point can be taken to be as follows.

Result 2: The small-graph condition for widespread percolation in a uni-partite graph of order N is:

$$\langle k \rangle^2 + \sum_k k(k-2)p_k > log(N) \quad . \qquad (7)$$

This can be understood as follows. If a graph contains a giant component, it is of order N and the size of the next largest component is typically $O(logN)$; thus, according to the theory of random graphs the margin for error in estimating a giant component is of order $\pm \log N$. In the criterion above, the criterion for a cluster that is much greater than unity is that the right hand side is greater than zero. To this we now add the magnitude of the uncertainty in order to reduce the likelihood of an incorrect conclusion.

The expression in (7) is not much more complex than the large-graph criterion. Moreover, all of our small-graph criteria retain their validity in the limit of large N. Hence we expect these small-graph criteria to be the most reliable choice for testing percolation in small systems. This expectation is borne out in the examples below.

From testing of the various criteria, the exact and statistical estimates are roughly comparable in their ability to detect percolation. The statistical tests we have examined are useful when only partial information about a graph is available.

Archipelago

Our reference implementation of the above criteria for testing node vulnerability and information flow, is a Java application program, with associated Perl scripts, which we call Archipelago. The name of the program is based on the whimsical association of our model of regions and bridges. An archipelago is a volcanic island that usually takes the form of a characteristic arc of tops jutting out of the water level. The tops look separate but are actually bridged just under water by a volcanic saddle. This is the form that arises naturally from organizing the visual layout of graphs according to centrality.

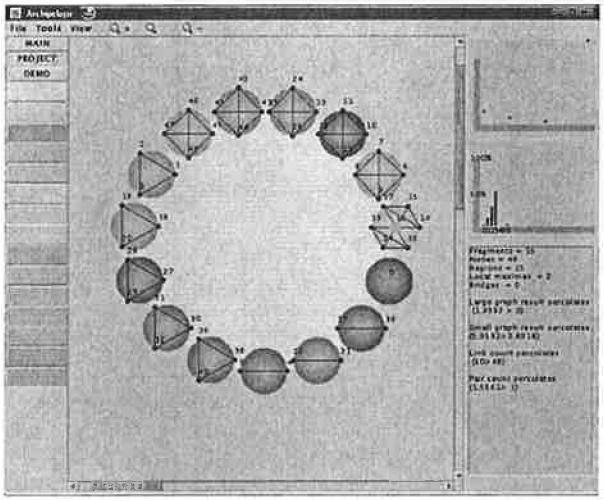


Figure 11: A scan of the student network at Oslo University College. This network is actually (in the absence of further links) quite secure against damage spreading, as it consists of many isolated pieces.

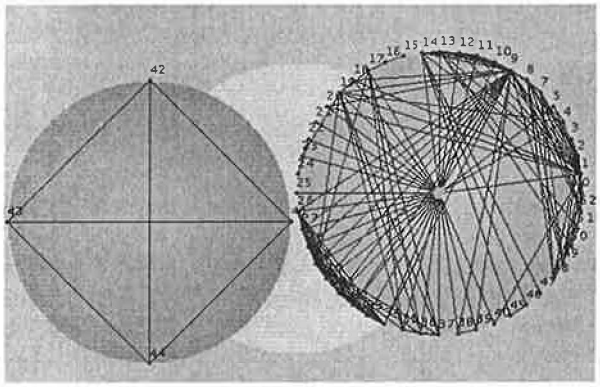


Figure 12: A scan of the staff network at Oslo University College. It is widely believed that this network is more secure than the student network, however this image shows otherwise. Since the staff are more trusting and more interconnected, the network is potentially far less secure.

The Archipelago application accepts, as input, an adjacency matrix of a graph. This can be entered manually or generated, e.g., by a Perl script that scans Unix file group associations. Archipelago calculates centrality and percolation criteria and organizes the regions into an archipelago of central peaks surrounded by their attendant nodes (colored in black). Nodes and links that act as bridges between the regions are colored red to highlight them, and disconnected fragments are colored with different background tints to distinguish them (see Figures 11 and 12).

The Application allows one to zoom in and move nodes around to increase the clarity of representation. One can also add and remove nodes and links to examine the vulnerability of the network to individual node removal, or spurious link addition.

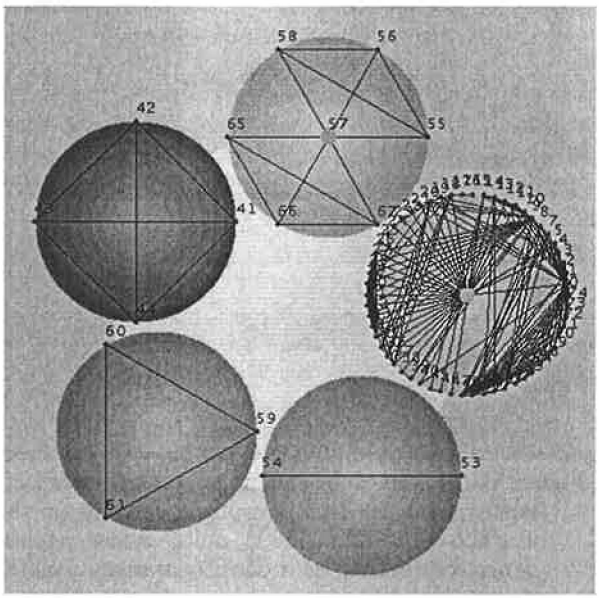


Figure 13: In fact the two graphs in Figure 20 and Figure 21 are not separate. Due to file sharing between staff and students, they are linked. When these links are take into account, the picture, even considering only file-sharing, becomes somewhat different. This shows how important it is to understand the boundaries of a system.

A database of information about the nodes is kept by the program, so that regular SQL searches can be made to search for covert links between users, based on common properties such as same family name, or same address. The addition of even a single covert link can completely change the landscape of a graph and make it percolate, or depose unstable centers.

Analyses in the right hand panel of the main window (Figure 11) show the histogram of the degree distribution in the graph and a log-log plot of the same, in order to reveal power law distributions that are considered to be particularly robust.

Potential Uses for Archipelago

We envisage several potential uses for this network analysis tool. We have already discussed some of these. Even armed with only centrality and percolation estimates, there is great flexibility in this mode of analysis.

Network Robustness

Determining how robust a network is to attack is an obvious task for the tool. Centrality determines the nodes that play the greatest role in the functioning of the system, and thus the obvious targets for attack. We can use Archipelago to identify these nodes and secure them from attack. Percolation, on the other hand, tells us that if an attack should succeed *somewhere*, what is the probability that it will lead to a significant security breach? Techniques like these have been applied to the spread of viruses like HIV in the world of medicine.

One result that is of future interest to network services is that from analyzing the Gnutella graph in Figure 10. Peer to peer technology has been claimed to be extremely decentralized and therefore robust: there is no central control, and hence no obvious point of attack. Our graph seems to contradict this notion, at first glance: it shows that the entire Gnutella file sharing network seems to split into two regions with two central peaks. However, these peaks are extremely shallow. One can use Archipelago to try 'taking out' these centers to see if the network can be broken up, and the spread of files curtailed. Attempting this has very little effect on the graph, however. The centers are barely above their neighbors, and the removal of a node simply makes way for a neighbor. The structure of the graph is almost immune to node deletions. That would not be the case in a hub-centralized network.

The same is not true of the other graphs, e.g., Figure 11. Here, taking out a central node of the largest fragment can cause the apparently tightly knit region to fall apart into more tenuously bridged, logically separate regions.

Resource Investment

In Figure 8, we considered how graphical analysis could be used to identify the places in a network where resources should be invested in order to maintain workflow. Here, a reorganization based on centrality illuminates the logical structure of the organization nicely. It consists of two regions: marketing and order processing, bound together by a human manager and a web server. The most central players in this web are the system administrator (who is relied upon by the staff and the servers), and the order processing department. The secure, continued functioning of this organization thus relies on sufficient resources being available to these two pieces of the puzzle. We see also an economic interpretation to the system that speaks of continuity in the face of component failure. ISO17799 considers this to be a part of systemic security, and we shall not argue.

Forensic Evidence

Finally, a speculative, but plausible way of using this analysis is in the solving of puzzles, crimes and other associative riddles. A forensic investigator is interested in piecing together various clues about who or what caused a breach of security. Such webs of clues are networks or graphs. The investigator would like to find the percolating fragments of a graph to see how associations link different persons together. It is not implausible that the most central nodes in such a graph would be key to solving the mystery, either as

places to look for more clues, or as masterminds behind the scheme.

One example, where this has been tested is in a case involving cheating on an electronic computer aided exam at OUC. Cheating was made possible by a failure of security in a server at an inopportune time. From printer logs it was possible to find out who had printed out a copy of the solutions to the exam during the test. From submissions of results and server logs, it was possible to match IP addresses to MAC addresses and student numbers and find out who was sitting close together during the test. From patterns of group project work, it was known which students generally worked together and would be likely to share the solution with one another. Finally, the submitted work, time and grades provided other correlations between students. The resulting network had a percolating component that implicated a group of students. It was later found that many of these were also connected as another kind of social subgroup of the class. Thus all of the clues resulted in the formation of a percolating web of 'suspects.'

The results, while not conclusive, provided a better than chance picture of who was likely implicated in the deception. Later information, from concerned students confirmed which of these were definite offenders and showed the central nodes in the network to be at the heart of the group who had originally printed out the solution. While this did not solve the crime completely, it pointed obvious fingers that made it possible to extend the analysis and learn much more than would otherwise have been possible.

We hope to go back to this kind of investigative work for further study and see whether it is possible to develop it into a truly useful tool.

Archipelago's database was intended for storing the kind of supporting information that could possibly lead to 'hidden links' being identified in graphs. For instance, if one searched the database for users who live at the same address, this would be a good cause to add a possible link to a graph, associating nodes with one another. Different strengths of connections between nodes could also be used to add a further level of gradation to the importance of connections. This added feature may be implemented easily, since it simply entails changing the binary entries of the adjacency matrix to real numbers, reflecting bond strengths.

Conclusions

We have implemented a graphical analysis tool for probing security and vulnerability within a human-computer network. We have used a number of analytical tests derived in [2]; these tests determine approximately when a threshold of free flow of information is reached, and localize the important nodes that underpin such flows.

We take care to note that the results we cite here depend crucially on where one chooses to place the boundaries for the network analysis. The methods will naturally work best when no artificial limits are placed on communication, e.g., by restricting to a local area network if there is frequent communication with the world beyond its gateway. On the other hand, if communication is dominated by local activity (e.g., by the presence of a firewall) then the analysis can be successfully applied to a smaller vessel.

At the start of this paper, we posed some basic questions that we can now answer.

1. *How do we identify weak spots in a system?* Eigenvalue centrality is the most revealing way of finding a system's vulnerable points. In order to find the true eigencenter of a system, one must be careful to include every kind of association between users, i.e., every channel of communication, in order to find the true center.
2. *How does one determine when system security is in danger of breaking down?* We have provided two simple tests that can be applied to graphical representations. These tests reveal what the eye cannot necessarily see in a complex system, namely when its level of random connectivity is so great that information can percolate to almost any user by some route. These tests can easily be calculated. The appearance or existence of a giant cluster is not related to the number of groups, but rather to how they are interconnected.

An attacker could easily perform the same analyses as a security administrator and, with only a superficial knowledge of the system, still manage to find the weak points. An attacker might choose to attack a node that is close to a central hub, since this attracts less attention but has a high probability of total penetration, so knowing where these points are allows one to implement a suitable protection policy. It is clear that the degree of danger is a policy dependent issue: the level of acceptable risk is different for each organization. What we have found here is a way of comparing strategies, that would allow us to minimize the relative risk, regardless of policy. This could be used in a game-theoretical analysis as suggested in [18]. The measurement scales we have obtained can easily be programmed into an analysis tool that administrators and security experts can use as a problem solving 'spreadsheet' for security. We are constructing such a graphical tool that administrators can use to make informed decisions [17].

There are many avenues for future research here. Understanding the percolation behavior in large graphs is a major field of research; several issues need to be understood here, but the main issue is how a graph splits into different clusters in real computer systems. There are usually two mechanisms at work in social graphs: purely random noise and node attraction – a 'rich get richer' accumulation of links at heavily

connected sites. Further ways of measuring centrality are also being developed and might lead to new insights. Various improvements can be made to our software, and we shall continue to develop this into a practical and useful tool.

Availability

Archipelago is available from Oslo University College http://www.iu.hio.no/archipelago .

Acknowledgement

GC and KE were partially supported by the Future & Emerging Technologies unit of the European Commission through Project BISON (IST-2001-38923).

Author Information

Tuva Stang and Fahimeh Pourbayat are final year bachelor students at Oslo University College; they are the principal eigenvectors of Archipelago. They can be contacted at TuvaHassel.Stang@iu.hio.no, and Fahimeh. Pourbayat@iu.hio.no for a limited time.

Mark Burgess is an associate professor at Oslo University College. He is the author of several books and of the system administration tool cfengine and is especially interested in mathematical methods in system administration. He can be contacted at mark@iu.hio.no.

Geoff Canright works with the Peer-to-peer Computing Group at Telenor R&D. He is interested in network analysis, self-organizing networks, and search on distributed information systems. He can be contacted at Geoffrey.Canright@telenor.com .

Kenth Engø works with the Future Com Business Group at Telenor R&D. He is interested in network analysis, risk analysis, and searching on networks.

Åsmund Weltzien works with the Peer-to-peer Computing Group at Telenor R&D. He is interested in (social) network analysis, innovation theory and spontaneous local network technologies.

References

[1] Burgess, M., *Principles of Network and System Administration*, J. Wiley & Sons, Chichester, 2000.

[2] Burgess, M., G. Canright, and K. Engø, "A Graph Theoretical Model of Computer Security: From File Access to Social Engineering," Submitted to *International Journal of Information Security*, 2003.

[3] Moser, L. E., "Graph Homomorphisms and the Design of Secure Computer Systems," *Proceedings of the Symposium on Security and Privacy, IEEE Computer Society*, p. 89, 1987.

[4] Williams, J. C., "A Graph Theoretic Formulation of Multilevel Secure Distributed Systems: An Overview," *Proceedings of the Symposium on Security and Privacy, IEEE Computer Society*, p. 97, 1987.

[5] Faloutsos, M., P. Faloutsos, and C. Faloutsos, "On Power-law Relationships of the Internet Topology," *Computer Communications Review*, Vol. 29, p. 251, 1999.

[6] Barabasi, A. L., R. Albert, and H. Jeong, "Scale-free Characteristics of Random Networks: Topology of the World-Wide Web," *Physica A*, Vol. 281, p. 69, 2000.

[7] Barabasi, A. L. and R. Albert, "Emergence of Scaling in Random Networks," *Science*, Vol. 286, p. 509, 1999.

[8] Albert, R., H. Jeong, and A. L. Barabasi, "Diameter of the World-Wide Web," *Nature*, Vol. 401, p. 130, 1999.

[9] Huberman, B. and A. Adamic, "Growth Dynamics of the World-Wide Web," *Nature*, Vol. 401, p. 131, 1999.

[10] Albert, R. and A. Barabási, "Statistical Mechanics of Complex Networks," *Rev. Mod. Phys.*, Vol. 74, 2002.

[11] Kao, M. Y., Data Security Equals Graph Connectivity, *SIAM Journal on Discrete Mathematics*, Vol. 9, Num. 87, 1996.

[12] Newman, M. E. J., S. H. Strogatz, and D. J. Watts, "Random Graphs with Arbitrary Degree Distributions and Their Applications," *Physical Review E*, Vol. 64, Num. 026118, 2001.

[13] Bonacich, P., "Power and Centrality: A Family of Measures," *American Journal of Sociology*, Vol. 92, pp. 1170-1182, 1987.

[14] Canright, G. and Å. Weltzien, "Multiplex Structure of the Communications Network in a Small Working Group," *Proceedings, International Sunbelt Social Network Conference XXIII*, Cancun, Mexico, 2003.

[15] Canright, G. and K. Engø, "A Natural Definition of Clusters and Roles in Undirected Graphs," *Paper in preparation*, 2003.

[16] Jovanovic, Mihajlo, *Private communication*, 2001.

[17] Burgess, M., G. Canright, T. Hassel Stang, F. Pourbayat, K. Engø, and Å. Weltzien, "Automated Security Analysis," *Paper submitted to LISA 2003*, 2003.

[18] Burgess, M., "Theoretical System Administration," *Proceedings of the Fourteenth Systems Administration Conference (LISA XIV)*, Usenix, Berkeley, CA, p. 1, 2000.

STRIDER: A Black-box, State-based Approach to Change and Configuration Management and Support

Yi-Min Wang, Chad Verbowski, John Dunagan, Yu Chen, Helen J. Wang, Chun Yuan, and Zheng Zhang – Microsoft Research

ABSTRACT

We describe a new approach, called Strider, to Change and Configuration Management and Support (CCMS). Strider is a black-box approach: without relying on specifications, it uses state differencing to identify potential causes of differing program behaviors, uses state tracing to identify actual, run-time state dependencies, and uses statistical behavior modeling for noise filtering. Strider is a state-based approach: instead of linking vague, high-level descriptions and symptoms to relevant actions, it models management and support problems in terms of individual, named pieces of low-level configuration state and provides precise mappings to user-friendly information through a computer genomics database. We use troubleshooting of configuration failures to demonstrate that the Strider approach reduces problem complexity by several orders of magnitude, making root cause analysis possible.

Introduction

Change and Configuration Management (CCM) refers to the task of monitoring configuration changes and maintaining systems in healthy configuration states. Change and Configuration Support (CCS) refers to the task of performing troubleshooting and repairs to bring systems back to healthy configuration states, after configuration failures have occurred.

CCMS of computer platforms with large install bases and large numbers of available third-party software packages have proved to be daunting tasks [LC01]. The large number of possible configurations and the lack of fully specified "golden states" [TH98] have made the problem appear to be intractable.

In particular, configuration problems caused by data sharing through persistent stores present a great challenge. Such shared stores may serve many purposes: they may contain system-wide resources that are naturally shared by all applications (e.g., the file system); they may allow applications installed at different times to discover and integrate with each other to provide a richer user experience; they may allow users to install new applications to customize default handlers or appearances of existing applications; they may allow individual applications to register with system services to reuse base functionalities; or they may allow individual components to register with host applications that provide an extensibility mechanism (e.g., toolbars in browsers).

Ideally, a white-box approach could greatly simplify the task: the developers of every OS component and every application would accurately and fully specify the set of configuration data that their programs use, the health invariants that subsets of these configuration data must satisfy, and the dependencies among the OS components and applications. Such information could then be used to compose machine-wide dependency information and golden configuration states.

In practice, studies have shown that [SC01, HD02] it is very difficult to declare and maintain accurate and conflict-free cross-application dependencies for every machine with a unique configuration. The problem is exacerbated by software upgrades/patches that create "forward dependencies" and by installation scripts that can perform tasks that have system-wide effects but are not captured by declared dependencies [HD02].

This motivates the need of black-box techniques to allow management and support of existing legacy applications, and to provide a smooth migration path to the ideal future of accurately and fully-specified systems. In this paper, we describe a new approach, called Strider, to achieve that goal. At the core of Strider is a *computer genomics database* that can accommodate specifications based on white-box knowledge as well as those derived from black-box experiments and discovery using state differencing (or *diffing*) and tracing techniques.

The main contributions of Strider are as follows, which also serve as the outline of the paper. First, we identify three *Strider principles* as the key to handling complexity in CCMS: *State-Based Analysis*, *Attack the Mess with the Mass*, and *Complexity-Noise Filtering*. Applying these principles allows us to decompose seemingly intractable CCMS problems into sub-problems, each of which is solved by a *Strider component*. Second, we introduce *Strider processes* as conceptual uses of various combinations of the Strider components to solve different problems, including troubleshooting, configuration certification, and change audit.

Third, we describe the *Strider toolkit* that implements the Strider components. Finally, we present the Strider troubleshooter that strings together components from the toolkit to implement the troubleshooting process. We evaluate the performance of the troubleshooter and discuss its limitations. To simplify our presentation, we will focus our discussion on a particular type of important configuration data – the Windows Registry [SR00], which provides hierarchical persistent storage for named, typed entries. The principles and techniques are generally applicable to other types of configuration stores and other platforms; we will discuss such applications at the end of the paper.

The Strider Principles

We begin by describing the three Strider principles, and use troubleshooting of configuration failures (i.e., errors resulting from mis-configuration) as the primary example to illustrate problem decomposition.

State-Based Analysis

A configuration failure occurs when a program modifies a piece of configuration data and, some time later, that same program or another program reads that modification and experiences a persistent failure that cannot be repaired by application restart or machine reboot. The failure can exhibit symptoms in the form of a program crash, program hang, error dialog box, or simply not delivering user-expected service.

Computer users (or support engineers) typically perform *symptom-based analysis* to troubleshoot configuration problems. Based on their knowledge and past experiences with similar problems, the users try to search the Web or a support-article database using search strings constructed in an ad-hoc way in an attempt to describe the symptoms. Such search is highly imprecise and often results in a large number of irrelevant articles. Furthermore, there is no guarantee that the repair actions suggested in these articles would actually modify the configuration data relevant to the failure in question.

In Strider, we propose *state-based analysis* as the primary approach to troubleshooting. Given a configuration failure, we represent it as a high dimensional state vector of all configuration data. For example, Windows XP machines typically have around 200,000 Registry entries; a configuration failure due to a faulty entry can be represented as a 200,000-dimensional vector that contains the entry. The main challenge is to narrow down the problem to that entry.

To reduce the dimensionality to the level that can be handled by humans, we develop mechanical techniques to exclude those entries that are irrelevant to the current failure, and develop statistical techniques to filter out those entries that are relevant but less likely to be the root cause. Once we narrow down the potential candidates to a small subset, we perform a precise lookup in a computer genomics database for each entry in the subset to identify potential fixes. Optionally, we can use the imprecise symptom descriptions at this later stage to help rank the importance of the candidates by matching the descriptions against information retrieved from the database.

Given the name of a Registry entry, the genomics database answers the following two questions:

1. *What is the function of this entry?* This provides any higher-level information that can help users understand the function of the entry. It may associate the entry with the application(s) or OS component(s) that are mainly responsible for updating it and therefore can potentially be used to correct problems caused by the entry. It may also identify the entry as "noise" from the viewpoint of configuration management because the entry is unlikely to cause configuration failures. (Labeling and filtering of noise will be discussed in more detail when we introduce the third Strider principle.)
2. *Are there known problems associated with this entry?* This quickly points to support articles on known problems caused by the entry, if any. Such information is useful for troubleshooting, but it can also be applied to correct Registry problems before they cause application failures.

The computer genomics database can be populated today through troubleshooting experiences and black-box experiments (e.g., we have recorded the Registry access traces of most of the Windows XP configuration actions), as well as through application-provided specifications in the future.

Attack the Mess With the Mass

Applying the first principle allows us to decompose the problem into three parts: mechanical, statistical, and database. We now develop the second principle to provide further decomposition of the mechanical part.

Every Windows XP machine starts with approximately 77,000 Registry entries from the CD installation process. The majority of users are given the freedom and flexibility to grow the Registry any way they want by configuring their machines differently and installing different sets of applications. Such freedom and flexibility helped create a large install base, but also created "*the mess*" – every machine has a unique configuration and applications on each machine can interact in a unique way. For example, the default handlers for file extensions or the behavior of an extensible browser may depend on the particular combination of software components installed on a system. When a configuration failure occurs, the lack of a golden state vector particular to the unique configuration at hand typically presents a major obstacle to troubleshooting.

In Strider, we make the observation that full-size, absolutely golden state vectors may not be necessary for CCMS problems. When a program fails due to a configuration problem on a particular machine at a

particular time, it suffices to find a state vector either from another machine or from the past on the same machine, where the program is/was working. In the space domain, "*the mass*" (i.e., the large install base) offers a high probability that one can find a healthy machine for cross-machine analysis. In the time domain, periodic state snapshot feature such as Windows XP System Restore [SR] can often provide a good state vector from the past for cross-time analysis.

Given a good state vector and a bad state vector, the mechanical part of Strider operates as follows. First, it performs a *state diffing* operation on the two vectors to obtain a sub-vector consisting of only the differences, which must capture the root cause for the difference in program behavior. Second, it asks the user to re-execute the failed program action and performs *state tracing* to record a sub-vector consisting of only those configuration data that are actually used as input to the current failed execution. Finally, it intersects the two sub-vectors to identify those that are potential root-cause candidates for the current failure. (An illustration of these three components is shown in Figure 1). Preliminary results from our experiments show that, since the diffing sub-vector and the tracing sub-vector are mostly orthogonal, the number of candidates in the intersection is often several orders of magnitude smaller than the full vector describing the bad state.

Once we have decomposed the mechanical part of Strider into state diffing, state tracing, and state intersection, we can utilize various combinations to further take advantage of the mass. For example, if a good state vector is available both from the past on the same machine and from one or more other machines, we can use multiple state diffing sub-vectors in the intersection to further reduce the size of the candidate set. Similarly, if the same application failure occurs on multiple machines, the state diffing and tracing sub-vectors from all these machines can be intersected together to further narrow down the candidate set. Even in situations where no state diffing sub-vector is available, intersecting multiple traces may help eliminate non-deterministic parts of execution traces due to other system activities and irrelevant to the deterministic application failure that is the troubleshooting target. A caveat is warranted here: in order for these more elaborate combinations to succeed, the root cause of the configuration failure must be a single entry or a fixed set of entries that differ between every sick/healthy pair. Fortunately, our experience has been that such root causes are indeed responsible for many configuration failures.

State diffing and state tracing distinguish Strider's black-box approach from the white-box approach: instead of relying on a *full specification* of *absolutely golden state* provided by software developers, we use state tracing to scope essentially a "*partial specification*" of the portion of configuration state actually accessed by the code path taken by the failed program execution, and use state diffing to take advantage of the "*good states relevant to this failure*" available in state snapshots from the past and/or from other machines where the program does not fail.

Complexity-Noise Filtering

An immediate concern about the mechanical part of Strider is that a large class of Registry entries are both updated and read frequently, which means that they will appear in both the diffing and tracing sub-vectors. Because of this, they will also appear in the intersection with high probability, and thus they will consistently inflate the size of the final candidate set. Timestamps,

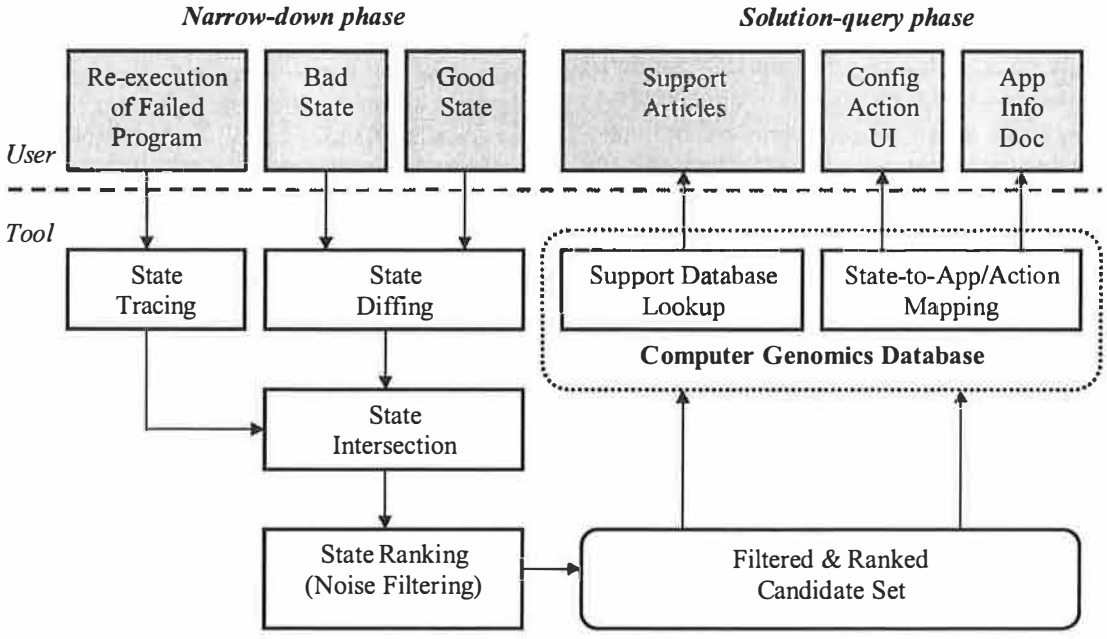


Figure 1: Strider process for troubleshooting.

usage counts, caches, seeds for random number generators, window positions, and MRU (Most Recently Used)-related information are such examples.

In Strider, we make the observation that such "high-frequency" entries should be considered "*operational states*" instead of the "configuration states" that we are mostly interested in for troubleshooting configuration failures. If a machine has been healthy in the presence of these high-frequency updates, then when a configuration failure occurs, these operational data are less likely to be the root cause. In contrast, configuration data that have not changed often in the machine's history but have changed recently since the application was last known to be working are more likely to be the root cause. This leads to the concept of *state ranking* based on *Inverse Change Frequency (ICF)*: we assign each candidate in the intersection a score that inversely depends on its change frequency, and prioritize the troubleshooting effort according to the score ranking; optionally, entries with scores below a threshold can be filtered out as noise and ignored. More sophisticated statistical analysis techniques that additionally take into account abnormal data content [G53, CG91] should further improve troubleshooting effectiveness.

The same observation can be applied to cross-machine analysis. Clearly, there are classes of Registry entries that always contain different data on different machines; for example, the data may be a function of computer names, user names, user security IDs, Globally Unique IDs (GUIDs), hardware IDs, IP addresses, etc. These entries constitute the "natural biological diversity" among machines and are less likely to be root causes of configuration failures. These differences are much like the human genes that are simply responsible for the natural diversity in human appearances and that are not thought to be the cause of any genetic disease even though they frequently appear as genetic differences between sick and healthy people.

In summary, the state-based approach starts with a large and complex problem: the Registry contains many entries, many of them changing. Fortunately, when we apply the complexity-noise filtering principle, these sources of complexity tend to filter themselves out, allowing us to focus on the fewer and simpler Registry entries that are most likely to be significant. This again distinguishes Strider's black-box approach from the white-box approach: instead of relying on a specification of operational data versus configuration data, we use behavior monitoring and statistical modeling to derive this distinction. Similar statistical techniques can also be used to predict potential failures by analyzing a large number of state vectors and flagging those that deviate from the "normal majority" as problematic ones that require special attention.

We note that, for any Registry entry that Strider filters out as noise, one can always construct a counterexample in which the entry is in fact the root cause; such a trade-off between false negatives and false positives is inherent in any statistical techniques. Our empirical results so far have indicated that noise filtering is essential for dealing with complexities and it allows successful troubleshooting of a large class of failures. We will discuss the limitations of noise filtering in more detail in a later section.

The Strider Processes

Applying the three Strider principles allows a decomposition of the troubleshooting problem into five Strider components: state diffing, state tracing, state intersection, state ranking, and the computer genomics database. In this paper, we use the term "Strider process" to refer to a conceptual use of some or all of the Strider components as building blocks in a specific way for a specific type of CCMS problem.

Figure 1 illustrates the Strider process for troubleshooting. In the narrow-down phase, the state diffing result and the failed application trace are intersected to produce a candidate set, which is then ranked and filtered by the state ranking module. As more and more troubleshooter reports are gathered, entries that are known to cause failures can be emphasized and entries that have repeatedly appeared in the reports as false positives can be de-emphasized.

In the solution-query phase, a genomics database lookup is performed for each entry in the candidate set, to yield one or more of the following three types of information: (1) support articles that describe known fixes of problems related to the entry; (2) user interface for performing configuration actions that can potentially correct the data content of the entry; and (3) information about the application that owns this entry.

We next describe two other Strider processes for different CCMS problems to demonstrate the flexibility provided by Strider's componentized approach. The "configuration certification" process addresses an important CCMS scenario. In this scenario, we would like to answer the question of whether an operational machine still conforms to a certified configuration and so is eligible for product support service. The Strider process would involve state diffing between the operational machine and a certified machine, followed by noise filtering of known entries unrelated to certification. State ranking with an adjustable threshold could provide a trade-off between the time spent in determining the conformance and the time wasted in providing support for non-conforming machines falsely determined to be in conformance. The genomics database could store information regarding commonly installed, unsupported hardware or software to speed up the determination of non-conformance.

Next we describe the "change audit" process. In the scenario targeted by this process, we would like to answer questions in the form of "*what has changed on*

my machine since last week?" This Strider process would involve always-on state tracing of all write operations with always-on noise filtering to control the size of the audit file. (State tracing captures the additional information of which process in what context made the changes, information which is typically not available from state diffing.) State ranking would distinguish significant configuration changes from the lesser ones. The genomics database would store mapping information that translates groups of changes into higher-level, user-friendly descriptions for better presentation.

The Strider Toolkit

We have implemented the full functionality of the first three Strider components in the Strider toolkit. A limited form of the state ranking component and part of the computer genomics database are also included in the toolkit.

The state diffing tool by default takes two System Restore checkpoints as input and produces an XML file containing Registry entries that exist in both checkpoints but have different data as well as those that exist in only one of the checkpoints. System Restore is a standard feature on Windows XP machines. It automatically saves a checkpoint of the Registry, selected files, and other configuration stores approximately every 24 hours. The number of available checkpoints depends on the maximum amount of disk space allocated for System Restore, which is set to 12% of each hard drive by default.

The tool also supports diffing of only selected Registry hives. For example, if a configuration failure occurs under one user account but does not occur under another user account on the same machine, then the root cause cannot reside in machine-wide Registry hives. Diffing only the per-user hives of the two users takes less time and reduces the number of false positives in the report.

The state tracing tool is implemented as a kernel-mode driver that, by default, intercepts and records every Registry call made by any application or OS component. It supports Include and Exclude filters for logging only those trace lines that contain or do not contain, respectively, specific sub-strings. It also supports efficient logging of only certain call types; for example, it can perform always-on logging of only write-related call types to provide a comprehensive change audit.

The state intersection tool uses a generic tree data structure to maintain a set of hierarchical names. It can take multiple state diffing files and/or multiple state tracing files as input. Each state entry in each of the input files is inserted into the tree and marked by the ID of its source file. Entries that are marked by the IDs of all input files are reported in the intersection result. As we extend the functionality of the state diffing and tracing tools beyond Registry to include other configuration stores such as files, application-specific XML configuration files, etc., the same data structure can be used to compute the intersection.

Ideally, on each machine running the Strider tool, the state ranking component should compute the ICF scores based on a customized "change frequency dictionary" for the local machine because each machine is configured and used in a different way and so the change behavior of configuration state may be different. Building such a customized dictionary at troubleshooting time would not be feasible because it would involve invoking the state diffing operation on every pair of consecutive checkpoints, which could take several hours (with five minutes per pair in today's implementation).

Currently, we include a "static dictionary" in the Strider executable and use the static scores in the dictionary for all state ranking operations. The dictionary was built from analyzing the change frequencies on the main desktop machine of one of the authors. Our troubleshooting experience so far has indicated that such a dictionary appears to be effective in ranking commonly updated Registry entries, but may miss many application- or machine-specific changes. We plan to replace it with another one built from multiple machines to increase its coverage and make it more representative. In the long run, we would like to have an always-on Windows service running on every machine, continuously updating a local, customized dictionary.

As an optimization, well-known Registry entries and sub-hierarchies that change very frequently and/or repeatedly appear as false positives in troubleshooting reports are filtered out in a keyword-based noise filtering step inserted right before the intersection; it is invoked after the intersection code reads an entry from an input file and before it inserts the entry into the tree structure. A second threshold-based noise filtering step is invoked after the intersection: it grays out entries with ICF scores below the (conservative) default threshold, which corresponds to a change frequency of 10% in the static dictionary. In addition to the ICF ranking, order ranking is also applied to assign more weight to entries that appear earlier in the trace, based on the intuition that later part of the trace may simply be a result of execution divergence caused by a bad value of an earlier entry.

Currently, part of the state-to-app/action mapping information of the genomics database is built into the executable. In a one-time experiment, we performed all commonly used Windows XP configuration actions and recorded their corresponding Registry update operations using the tracing tool. The reverse mappings can then be used to provide state-to-action mappings at troubleshooting time. As more state-to-app mapping information is obtained through experiments and actual troubleshooting experience, we plan to build a Web service for entering and querying such information. The same Web service will also be used to implement the support article lookup part of the genomics database, which is currently compiled as a list on a Web page with pointers into a trouble-ticket database and a support article database.

Experimental Results

Clearly, the Strider approach would not work if the following worst case were the norm: a large percentage of the Registry entries change every day and a large percentage of them are used by every application action, resulting in a large candidate set that no human could reasonably handle.

We present empirical results in this section to show that the above worst case is not the norm. We first present measurements of Registry change frequencies from five machines to study the typical size of the state diffing set. Then we present results from troubleshooting experiments to evaluate the effectiveness of additional state tracing, intersection, and ranking.

We use the ten cases listed below in our experiments. They were all real-world failures that troubled some users. To allow parameterized experiments, we reproduced these failures on machines in our group and ran Strider to produce the results. We used configuration user interface (e.g, Control Panel applets) to inject the failures whenever possible, and used direct editing of the Registry for the remaining cases. All the chosen machines were desktop machines used by their owners on a daily basis. This is important because they would exhibit "regular" Registry change behaviors; using test machines from our lab (that have little installation/configuration activity) would have produced better but invalid results. We also study the sensitivity of the results with respect to the choice of machines to inject the failures. Preliminary results from cross-machine troubleshooting are also discussed.

1. Systems Restore: no available checkpoints are displayed because the calendar control object cannot be started due to a missing Registry entry.
2. JPG: right-clicking on a JPG image and choosing the *Send To → Mail Recipient* option no longer offer the resize option dialog box due to a missing Registry entry.
3. Outlook: user is always asked upon exiting Outlook whether she wants to permanently delete all emails in the Deleted Items folder, due to a hard-to-find setting.
4. Printing: printing to a duplex-named printer always produces single-sided printing, due to a hard-to-find setting.
5. IE Passwords: Internet Explorer (IE) browser no longer offers to automatically save passwords; the option to re-enable the feature is difficult to find.
6. Media Player: Windows Media Player "*Open URL*" function would fail if the EnableAutodial Registry entry is changed from 0 to 1 on a corporate desktop.
7. IM: MSN Instant Messenger (IM) would significantly slow down if the firewall client is disabled on a corporate desktop.
8. IE Proxy: IE on a machine with a corporate proxy setting would fail when the machine is connected to a home network.
9. IE Offline: IE "*Work Offline*" option may be automatically turned on without user knowledge; user would then be presented with a cached offline page instead of the default start page when launching IE.
10. Taskbar: IE windows would be unexpectedly grouped under the Windows Explorer taskbar group, due to the addition of a Registry entry.

Registry Change Behavior

The common perception of the Windows Registry is that it contains an enormous amount of undocumented configuration information that is accessed frequently by various applications and OS components. To our knowledge, the study that we present in this section is the first quantitative study of Registry change behavior. In addition to providing insights for the troubleshooting problem, the study should serve as a useful guide for the general CCM community as well.

We studied the Registry from two perspectives. First, we looked at the aggregate change behavior of the Registry over a long period of time, ranging from 77 to 84 days. (These numbers are roughly determined by the number of available System Restore checkpoints per machine.) Next, we looked at the daily behavior of the Registry over the same observational period. We expect that Strider troubleshooting will most frequently be applied to a good checkpoint and a bad checkpoint that are close together in time, and therefore we expect the daily behavior of the Registry to be a good guide to the performance of the state diffing part of the Strider toolkit.

The machines in our study consisted of four developer workstations and one knowledge worker's machine, each of them in daily use. Figure 2 shows the Registry change statistics we observed across the five machines over the entirety of the observational periods. We present the number of Registry values at the end of the period for each machine – these vary from just under 140,000 to almost 240,000.

Machine	Number of Registry Values	Days Observed	% Never Changed	% Operational	% Remaining
1	139,458	84	95.3%	2.6%	2.1%
2	213,574	84	90.4%	1.9%	7.7%
3	232,890	84	89.6%	5.6%	4.8%
4	237,622	77	79.3%	1.2%	19.5
5	200,812	84	86.8%	1.9%	11.3%

Figure 2: Registry change statistics.

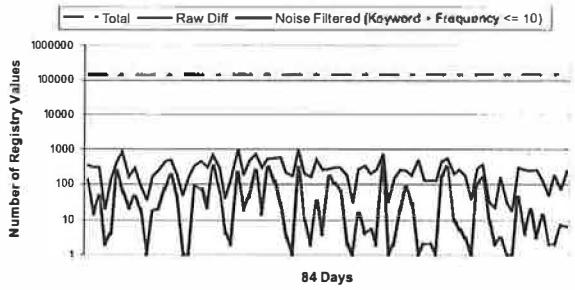


Figure 3a: Machine #1's Registry changes over time.

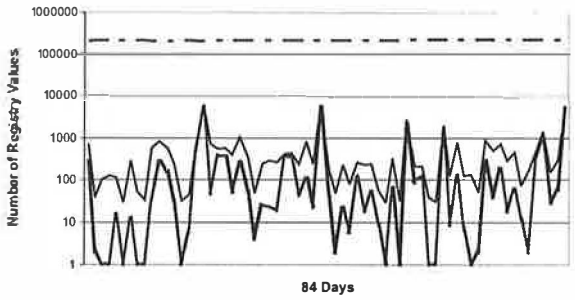


Figure 3b: Machine #2's Registry changes over time.

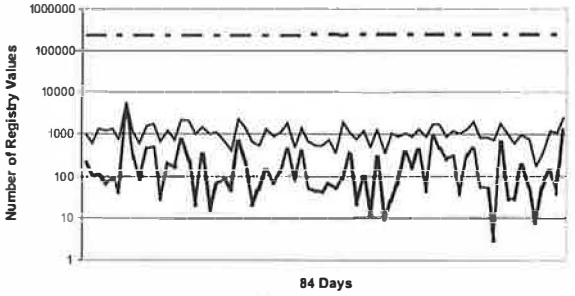


Figure 3c: Machine #3's Registry changes over time.

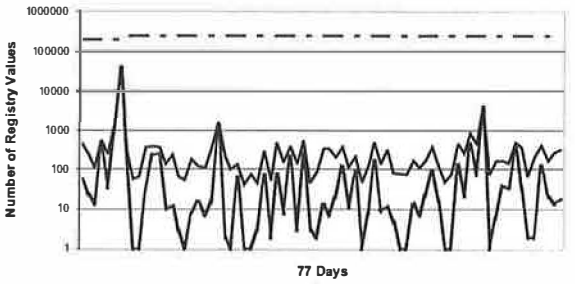


Figure 3d: Machine #4's Registry changes over time.

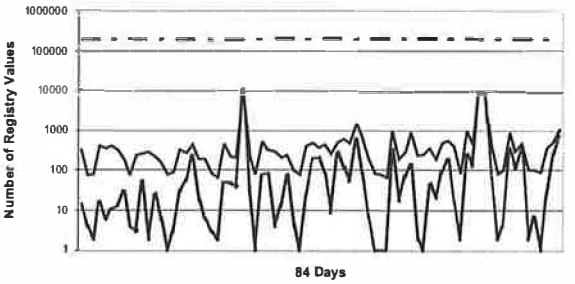


Figure 3e: Machine #5's Registry changes over time.

Figure 3: Registry daily changes with and without noise filtering.

On four of the five machines (#1, #2, #3 and #5), only 4.7%-13.2% of the Registry ever changed. Applying the noise filtering techniques (i.e., keyword-based filtering and change-frequency threshold-based filtering by excluding any Registry entry that changed more than 10 times during the period) to these machines yielded that the number of Registry changes that were classified as non-operational and so potentially configuration-related ranged from 2.1% to 11.3%.

On machine #4, 20.7% of the Registry changed. Looking at this machine's history in more detail, we found that the majority of the changes were due to a single large installation: on this one day, 16% of the Registry changed. If we exclude this single large installation when calculating the Registry change percentage, we find that the changes drop to around 4% of the total Registry size.

This number suggests that if we needed to troubleshoot this machine, state diffing of any pair of checkpoints on the same side of the large installation (i.e., either both before or both after) would likely result in a number of potentially significant entries that is comparable to the number found by state diffing on one of the machines with a small change percentage over the entire period. If the state diffing period must cover the large installation, then we need to rely on the intersection component to reduce the complexity, which will be discussed shortly.

Now we turn our attention to the daily behavior of the Registry. Figure 3 illustrates the daily behavior across all five machines. Because checkpoints may be taken for multiple reasons (by the users manually, by System Restore-aware installers prior to installations, or by System Restore service periodically), we were careful to ensure that we only included one checkpoint per 24-hour bucket in our analysis. Therefore the diff sizes shown in Figure 3 correspond to a gap of slightly more than 24 hours on average. The spike in machine #4's diff size due to the single large installation (mentioned in the previous paragraph) is clearly visible near the beginning of the observational period.

Across all five machines, the median number of changing Registry values on any given day is 302. After applying the Strider noise filtering, the median number drops to only 29. This demonstrates the additional power of noise filtering when applied to changes between checkpoints taken on consecutive days. This has the following simple explanation: although the percentage of operational Registry entries as shown in Figure 2 may seem low (between 1% and 6%), these entries changed frequently and so appeared much more often in daily diff results. Noise filtering effectively identifies these entries, which comprise a large portion of any daily diff, as unlikely to reflect significant configuration changes.

Same-machine, Cross-time Troubleshooting

Troubleshooting Effectiveness

Figure 4 (a) and (b) present our experimental results on the effectiveness of Strider troubleshooting

for the 10 cases, all with checkpoints that are approximately seven days apart. Along the horizontal axis,

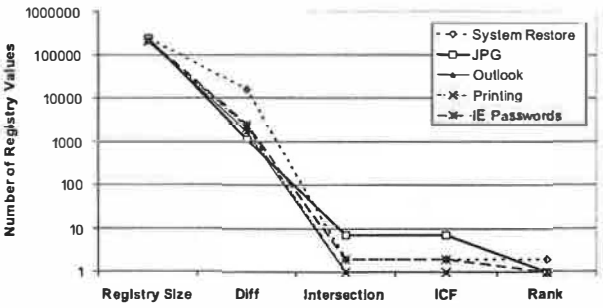


Figure 4a: Cases #1 to #5.

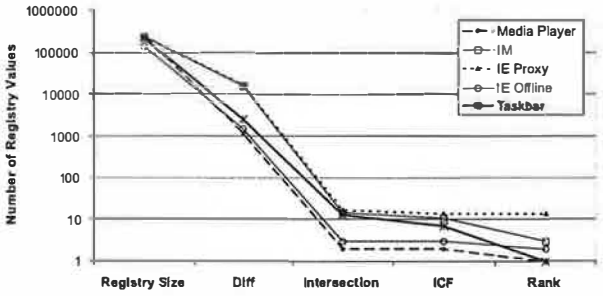


Figure 4b: Cases #6 to #10.

Figure 4: Same-machine, cross-time troubleshooting effectiveness.

"Registry Size" is the average number of Registry values of the two checkpoints; "Diff" is the number of Registry values in the state diffing result; "Intersection" is the total number of Registry values appearing in the report, which consist of all the entries in the intersection of keyword-filtered diff and trace; "ICF" excludes those entries in the report whose ICF scores are below the default threshold; "Rank" is the order ranking of the root cause in the ICF-filtered list.

The effect of each step in the Strider troubleshooting process is evident from the Figures. Typically, state diffing reduces the dimensionality by two orders of magnitude (from 200,000 to roughly around 2,000) and diff-trace intersection reduces it by another two orders of magnitude (from 2,000 to below 20). Even in the three cases where state diffing could provide only one-order-of-magnitude reduction because the 7-day period covered some significant installation events, the intersection still effectively brought down the number of candidates to below 20.

The ICF threshold-based noise filtering provided additional help for the three cases with more than 10 entries in the intersection (in Figure 4 (b)): it reduced the numbers from 17, 15, and 13 to 14, 11, and 7, respectively. The final ranking summarizes the overall effectiveness of Strider troubleshooting: the actual root cause was identified as the number 1 candidate in 6 of the 10 cases, as number 2 in two cases, and as number 3 in one case. The root-cause rank for the IE Proxy case was 14, which would require more manual analysis effort to filter out the false positives. We are currently investigating ways to group together relevant final entries to aid manual analysis.

Sensitivity Analysis

We performed additional experiments to study the sensitivity of the troubleshooting results to variation in the machine being examined and the time interval of the diff. We let the time between the good checkpoint and the bad checkpoint vary between 3, 7, or 14 days. We varied the machine under consideration across all five machines in our study, and we

	System Restore (case 1)		JPG (case 2)		Media Player (case 6)		IM (case 7)	
Machine #1	3 days	1	3 days	1	3 days	1	3 days	1
	7	1	7	1	7	1	7	1
	14	1	14	1	14	1	14	1
Machine #2	3 days	2	3 days	1	3 days	1	3 days	1
	7	2	7	1	7	1	7	1
	14	2	14	1	14	1	14	1
Machine #3	3 days	2	3 days	1	3 days	2	3 days	3
	7	2	7	1	7	2	7	3
	14	2	14	1	14	2	14	3
Machine #4	3 days	2	3 days	1	3 days	1	3 days	N/A
	7	2	7	1	7	1	7	N/A
	14	2	14	1	14	1	14	N/A
Machine #5	3 days	2	3 days	1	3 days	1	3 days	1
	7	2	7	1	7	1	7	1
	14	2	14	1	14	2	14	1

Figure 5: Sensitivity analysis of same-machine, cross-time troubleshooting (numbers are final root-cause ranks).

examined four cases: System Restore (case 1), JPG (case 2), Media Player (case 6), and IM (case 7). The final ranking results are presented in Figure 5.

We found the Strider troubleshooter to be robust to both factors being studied, although varying the factors did have some impact. In three of the four cases, the choice of machine affected the root-cause ranking, although the final rank remains number 3 or better in every case. In the case of machine #5 with case 6, we found that varying the offset in time from 7 to 14 days caused the root-cause rank to drop from 1 to 2.

Cross-machine Troubleshooting

Although the current version of the Strider toolkit is primarily targeted at same-machine, cross-time troubleshooting, we have conducted some preliminary experiments and found that it can be useful for cross-machine troubleshooting as well. We used the same 10 cases, but with checkpoints from two different machines: the configuration failure was introduced into one machine to make the target program action fail, while the same action succeeded on the other one.

Figure 6 shows the results. First, we observe that the current state diffing tool is less effective in the cross-machine scenario; it reduced the number of entries by about two thirds, in contract with the two orders of magnitude in the same-machine case. There are at least two factors that contributed to this: (1) different machines can simply have very different sets of installed programs; (2) the "same" Registry entries can appear to be different on different machines because their names contain machine-specific information. We are currently investigating a set of mapping rules to eliminate the latter.

Fortunately, the intersection operation remained effective and reduced the number to below 100 in all cases. The ICF noise filtering is only slightly useful for half of the cases because the static dictionary built from cross-time diffing analysis may not be suitable for the cross-machine scenario. We expect that a separate dictionary based on cross-machine diffing analysis of each Registry entry among a large number of checkpoints would improve the filtering.

The final step of applying the order-ranking heuristics was still mostly effective: in eight of the 10 cases, the root cause ranked number 10 or better. But for the IM case and the IE Proxy case, the root cause ranked 36 and 33, respectively.

Discussions and Future Work

In this section, we discuss additional issues and factors that can potentially impact the effectiveness of Strider troubleshooting and were not covered by our performance evaluations presented in the previous

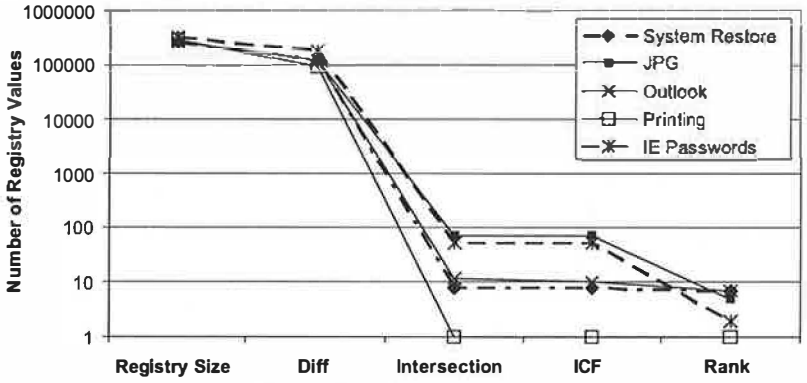


Figure 6a: Cases #1 to #5.

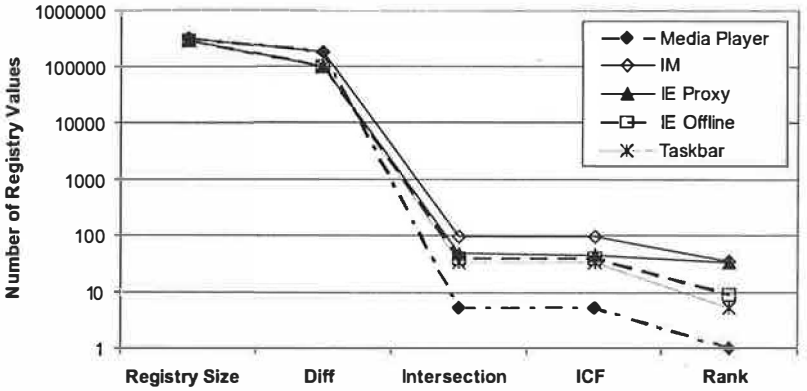


Figure 6b: Cases #6 to #10.

Figure 6: Preliminary results of cross-machine troubleshooting effectiveness.

section. We also discuss several types of problems for which the current version of Strider cannot successfully provide diagnosis, and we outline our future work directed at addressing these problems. In general, the challenge is to ensure that the mechanical operations capture the root cause, to understand the limitations of our current noise filtering techniques, and then to further exploit the mass to facilitate the final step of root cause analysis.

Capturing The Root Cause

In most cases, it is fairly clear which application's execution should be traced. For example, in the Media Player case, we traced wmplayer.exe; in the IM case, we traced msmsgs.exe. Per-process traces were used for all the cases except for the "Taskbar" case, in which traces for both iexplore.exe (IE) and explorer.exe (Windows Explorer) were included because it was difficult to determine from the symptom which one was the offending application.

Although they usually achieve good root-cause ranking, per-process traces capture only "direct dependencies" (i.e., Registry accesses made by the target process) and may miss the root cause contained in "indirect dependencies." For example, in a case where a pop-up stopper designed to stop pop-up ads interfered with the normal operations of a Web site, the root-cause Registry entry was accessed from a separate process, rather than from the browser process itself. We plan to enhance Strider with process dependency tracking so that it can capture indirect dependencies without resorting to using all-process traces.

In addition to indirect dependencies, "asynchronous dependencies" pose another challenge. Strider implicitly assumes that the root-cause entry must be accessed synchronously during the user-selected time interval for state tracing. However, it is possible that the traced application had read the root-cause entry before the tracing was started. For example, while in most of the 10 cases the application actions that the user should trace were well-defined and did contain the root causes, the remaining cases required tracing of application launching as well as the application action that led to the observed failure.

To study the effect of less-experienced Strider users always using the longer traces (i.e., since application launch) to avoid missing the root cause, we performed further experiments by replacing the action-only traces with the longer traces in Cases 1, 3, 4, 5, and 6. This would drop the root-cause ranking from (2, 1, 1, 1, 2) to (3, 12, 1, 9, 17), respectively, which is still acceptable but may require significantly more troubleshooting effort depending on whether the additional false positives are easy to filter manually. Our long-term direction is to develop efficient, always-on tracing and logging to relieve users of the responsibility to specify when to start and stop tracing.

Similarly, users may specify incorrect good states due to either incorrect memory or latencies between state corruption and application failure. This would cause the state diffing results and thus the intersection to miss the root cause. A near-term solution is to encourage users to be conservative in selecting good states. Our long-term direction is to develop statistical techniques that automatically analyze multiple good and bad checkpoints to relieve users of the burden.

Limitations of Noise Filtering

As discussed previously, statistical techniques such as Inverse Change Frequency ranking for filtering out false positives naturally introduce the possibility of false negatives. Although it is difficult to provide conclusive arguments that noise filtering does not introduce significant false negatives without a large number of failure cases, our experience has shown that it works well in practice and has allowed successful root cause analyses of tens of cases.

We now describe several types of problems that could potentially defeat Strider's current noise filtering strategy. Our plan is to refine the filtering rules as we encounter false negatives, and revisit the design if we gain concrete evidence that a significant number of real-world root causes actually fall into the false-negative category.

- **Usage counters**: for example, the behavior of a trial software package may change when its usage count exceeds a certain threshold.
- **Window positions**: for example, a corrupted entry that is supposed to remember the last position of an application window may cause display problems.
- **MRU and cache-related information**: for example, "last server connected" may be the root cause of a client application failure; "last file opened" may cause a document processing program to fail upon launch; a cached Web page may cause undesirable behavior in a browser.
- **Per-session data**: some application data may be updated on a per-session basis and have dependencies on the current environment; failures may occur when a user tries to restore such per-session data. Current Strider noise filtering would have mistakenly filtered out such data.
- **Data coupling**: a single Registry entry may contain both operational data and configuration data. For example, we have encountered a case where Word was used as the default email editor for Outlook and a certain document navigation option could not be turned off. The option was, unfortunately, controlled by some Registry entries containing binary blobs of data, and these binary blobs apparently contained operational data as well and so had low ICF scores. These entries were incorrectly filtered out as noise originally (i.e., grayed out in the report), but later determined to contain the root cause through further investigation. Once a false negative is

discovered, the change frequency dictionary built in to the Strider executable is updated to assign the entry a very high score, reflecting the fact that the entry has been identified as the root cause of an actual configuration failure.

Exploiting the Mass

The four orders of magnitude in dimensionality reduction typically achieved by the mechanical steps of Strider was a significant starting point for us to handle the complexity of Registry problems. However, we have encountered cases in which ICF noise filtering and order ranking failed to offer the final reduction of another order of magnitude and the users were left with tens of candidates to investigate. In some cross-machine cases, the final reports still contained hundreds of candidates and root cause analysis remained very difficult.

We plan to address this challenge by further exploiting the mass. We are collecting a large number of Registry snapshots in our "GeneBank" and plan to generalize the diff-based techniques to statistical analyses across multiple snapshots. In particular, root-cause candidates containing data that clearly deviate from the "normal majority" will be ranked higher. We are also enhancing the tracing and noise-filtering techniques to enable always-on logging and analysis on a large number of machines for building and reporting known-good behavioral models.

Beyond the Windows Registry

Although we have focused on troubleshooting Windows Registry-related problems, the Strider techniques are generally applicable to any shared, persistent configuration store on any operating system platform.

We are currently extending the Strider implementation to provide troubleshooting of configuration problems due to changes in files and directories/folders. The implementation will utilize the file-change log information from System Restore to detect which files have been changed, use a filter driver to trace which files are being accessed as part of an application action, reuse the tree structure for computing the intersection, use file change frequency for state ranking, and rely on information from the genomics database to identify directories containing temporary files as known noise, to provide mappings of which files belong to which applications or OS components, and to point to support articles documenting known problems with certain files.

Similar techniques can also be applied to Unix machines. Unix configuration generally appears in files under /etc/. For example, user account information is in /etc/passwd, the IP address of the DNS server is in /etc/resolv.conf, and the many parameters for the X server are typically in /etc/X11/. Although many configuration files are used by a single program or OS facility, quite a few well-known configuration files are shared by multiple programs and so are subject to similar configuration problems as those in the Window Registry.

The most notable example is /etc/mailcap, which contains a system-wide mapping from MIME types to commands for handling them. Any software that displays or edits a particular type of MIME file may want to install entries in the mailcap file, so that other programs can use it to display or edit that file type. Therefore, applications that can handle common file types could write conflicting entries into the mailcap configuration file.

Another example is /etc/inetd.conf. Rather than having a separate daemon for each type of connection (for example, finger, telnet, rlogin, rsh, smtp, ftp) listening on its own port for incoming connections, the meta-daemon inetd listens on all the ports and starts an instance of the appropriate daemon on demand as each connection comes in. Each of the individual daemons is required to add an entry to inetd.conf when it is installed and remove that entry upon uninstallation. If two daemons use the same port, their entries in inetd.conf may conflict [HD02].

In addition, ill-written Unix applications may modify the environment variables in a user's .cshrc configuration file for their own operations. Such practices may result in conflicts in environment variables. When these conflicts result in faulty application executions, users typically resort to application reinstallation to repair the problem. Strider troubleshooting can help identify the root cause to potentially provide a less disruptive repair and avoid future occurrences of the same problem.

Related Work

The body of work related to systems management through specification is quite large [B95, RPM97, CG99, O00, KE02, TRI]. The general approach is to provide languages and tools to allow developers or system administrators to specify "rules" of proper system behavior and configuration for monitoring, and "actions" to correct any detected lack of compliance with a given rule to enable the system to converge with the specified requirements. Strider complements the specification-based approach by adopting a black-box approach to discover unspecified rules of proper system operation and gradually build up a genomics database of known-good requirements and known-bad issues.

Gossips [GWO01] provides an extensible, object-oriented framework for monitoring distributed systems in an IT-environment. Each gossips process running on a participating client gathers and analyzes system state-related data, and reports any interesting state-changes to a central server. Although Gossips also maintains a knowledge base of known problems indexed by state-related information, the "states"

refer to the condition of a system or service (such as *working/broken*), which are quite different from the lower-level, more precise "configuration data" states in the Strider genomics database.

In a recent position paper, Redstone, et al. [RSB03] described a vision of an automated problem diagnosis system that automatically captures aspects of a computer's state, behavior, and symptoms necessary to characterize the problem, and matches such information against problem reports stored in a structured database. In the Strider project, we have focused on developing actual root cause analysis technologies for configuration failures by using state diff and trace information to characterize them. Symptom descriptions are used as secondary information and are still provided by the user because many Registry-related "problems" can only be defined against user expectation. An earlier version of Strider provided automatic search of support database for high-ranking root-cause candidates [WVS03]; but we have observed that such an approach can only be effective after a large number of support articles are written in a structured, machine-readable format and stored in the genomics database.

The concept of problem identification as deviant behavior from a "normal majority" by applying statistical techniques to a large number of samples has emerged in several areas in recent years. Engler, et al. [ECH+01] described techniques that automatically extract correctness rules from the source code itself (rather than the programmers) and flag deviations, and that use statistical ranking to prioritize the inspection effort. Liblit, et al. [LAZ+03] proposed a sampling infrastructure for gathering information about a large number of actual program executions experienced by a user community, based on which predicate guessing and elimination are used to isolate deterministic bugs and statistical modeling is used to isolate nondeterministic errors by identifying correlation between behaviors and failures.

Apap, et al. [AHH+02] presented a host-based intrusion detection system that builds a model of normal Registry behavior through training and showed that anomaly detection against the model can identify malicious activities with relatively high accuracy and low false positive rate. The PinPoint root cause analysis framework [CKF+02] applies data clustering analysis to a large number of multi-tier request-response traces tagged with perceived success/failure status to determine the subset of components that are most likely to be the cause of failures.

Summary

We have proposed the Strider state-based approach to change and configuration management and support, and built and evaluated a system based on this approach. The approach allows a decomposition of complex problems into five Strider components that can be used as building blocks in various scenarios. In the context of our primary example, troubleshooting of configuration failures, we have demonstrated that combining the black-box techniques of state differencing, tracing, intersection, and ranking can effectively narrow down the list of root-cause candidates for many real-world cases. As we continue to build up the computer genomics database, where we provide precise mappings from configuration state items to their known functions and/or problems, more knowledge will be captured in a structured format, enabling even more effective root cause analysis. Our future work includes providing differencing and tracing of more types of configuration state to increase coverage, collecting a large number of state snapshots and program traces to enable advanced statistical analysis and reduce Strider's dependence on manual steps, and evolving the current Strider toolkit for troubleshooting into a systems management framework for self-monitoring and self-healing.

Acknowledgement

We would like to express our sincere thanks to our shepherd Alva L Couch for his valuable feedback, to Jidong Wang and Ji-Rong Wen for recording the UI-to-Registry mapping data, to those colleagues who provided Registry snapshots for change behavior analysis, and to those people who contributed the configuration failure cases for our experiments.

Author Information

Yi-Min Wang is the project lead of Strider. He has been with the Systems and Networking Group at Microsoft Research in Redmond since 1998. He received his Ph.D. in Electrical and Computer Engineering from University of Illinois at Urbana-Champaign in 1993, and worked at AT&T Bell Labs from 1993 to 1997. His research interests include systems management, fault tolerance, and distributed systems.

Chad Verbowski has been a Development Lead at Microsoft for the past five years working on multiple Windows OS components and Systems Management software. Previously Chad worked for several years building and deploying management systems in real world environments. He is currently with Microsoft Research. Chad can be reached at chadv@microsoft.com.

John Dunagan is currently a member of the Systems and Networking Group at Microsoft Research. From 1998 to 2002, he was a member of MIT's Laboratory for Computer Science. He received his Ph.D. in Mathematics from MIT in 2002.

Yu Chen graduated from Peking University in 2001 with a Master degree in Computer Science. He joined Microsoft Research Asia and has been working in the Media Management Group. He can be reached at ychen@microsoft.com.

Helen J. Wang is a researcher in the Systems and Networking group at Microsoft Research, Redmond. Her research interests are in networking, protocol architectures, security, system management, mobile/wireless computing, and wide-area large scale distributed system design. She received her Ph.D. in Computer Science from U. C. Berkeley in 2001.

Chun Yuan is an Associate Researcher with Microsoft Research Asia. He received his Ph.D. in Computer Science from the University of Science and Technology of China in 2001. His research interests include peer-to-peer systems.

Zheng Zhang is Project Lead and Researcher in Microsoft Research Asia and was with HP-Lab as Member of Technical Staff from 1996 to 2001. He obtained his Ph.D. from University of Illinois at Urbana-Champaign. His current research interests focus on self-organized distributed system.

References

[AHH+02] Apap, F., A. Honig, S. Hershkop, E. Eskin, and S. J. Stolfo, "Detecting Malicious Software by Monitoring Anomalous Windows Registry Accesses," *Proc. of the Fifth International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2002.

[B95] Burgess, M., "A Site Configuration Engine," *Computing Systems*, Vol. 8, p. 309, 1995.

[CG91] Church, K. W. and W. A. Gale, "A Comparison of the Enhanced Good-Turing and Deleted Estimation Methods for Estimating Probabilities of English Bigrams," *Computer Speech and Language*, Vol. 5, pp. 19-54, 1991.

[CG99] Couch, Alva and M. Gilfix, "It's Elementary, Dear Watson: Applying Logic Programming to Convergent System Management Processes," *Proc. of LISA*, 1999.

[CKF+02] Chen, M., E. Kiciman, E. Fratkin, A. Fox, and E. Brewer, "Pinpoint: Problem Determination in Large, Dynamic, Internet Services," *Proc. Int. Conf. on Dependable Systems and Networks (IPDS Track)*, 2002.

[DG01] Dennis, C. and R. Gallagher, *The Human Genome*, Nature Publishing Group, 2001.

[ECH+01] Engler, D., D. Y. Chen, S. Hallem, A. Chou, and B. Chelf, "Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code," *Proc. ACM Symp. on Operating Systems Principles*, October, 2001.

[G53] Good, I. J., "The Population Frequencies of Species and the Estimation of Population Parameters," Biometrika, Vol. 40, pp. 237-264, 1953.

[GWO01] Götsch, V., A. Wuersch, and T. Oetiker, "Gossips: System and Service Monitor," *Proc. of LISA*, 2001.

[HD02] Hart, J. and J. D'Amelia, "An Analysis of RPM Validation Drift," *Proc. LISA*, 2002.

[KE02] Keller, A. and C. Ensel, "An Approach for Managing Service Dependencies with XML and the Resource Description Framework," *Journal of Network and Systems Management*, Vol. 10, Num. 2, June, 2002.

[LAZ+03] Liblit, B., A. Aiken, A. X. Zheng, and M. I. Jordan, "Bug Isolation via Remote Program Sampling," *Proc. Programming Language Design and Implementation (PLDI)*, pp. 141-154, 2003.

[LC01] Larsson, M. and I. Crnkovic, "Configuration Management for Component-based Systems," *Proc. Int. Conf. on Software Engineering (ICSE)*, May 2001.

[O00] Osterlund, R., "PIKT: Problem Informant/Killer Tool," *Proc. LISA*, 2000.

[RPM97] Bailey, E., *Maximum RPM*, 1997.

[RSB03] Redstone, J. A., M. M. Swift, B. N. Bershad, "Using Computers to Diagnose Computer Problems," *Proc. HotOS*, 2003.

[SC01] Sun, Y. and A. L. Couch, "Global Analysis of Dynamic Library Dependencies," *Proc. of LISA*, 2001.

[SR] *Windows XP System Restore*, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnwxp/html/windowsxpsystemrestore.asp .

[SR00] Solomon, D. A. and M. Russinovich, "Inside Microsoft Windows 2000," Microsoft Press, 3rd edition, Sept., 2000.

[TH98] Traugott, S. and J. Huddleston, "Bootstrapping an Infrastructure," *Proc. LISA*, 1998.

[TRI] *Tripwire*, http://www.tripwire.com/ .

[WVS03] Wang, Y. M., C. Verbowski, and D. R. Simon, "Persistent-state Checkpoint Comparison for Troubleshooting Configuration Failures," *Proc. Int. Conf. on Dependable Systems and Networks (DSN)*, 2003.

CDSS: Secure Distribution of Software Installation Media Images in a Heterogeneous Environment

Ted Cabeen – Impulse Internet Services
Job Bogan – Consultant

ABSTRACT

CDSS is a framework for the distribution of software installation media images and their contents over multiple file sharing protocols. The CDSS system provides a unique isolated server instance for every accessing user, even when another instance of that server is already running. CDSS uses the Linux host-based firewall system to transparently redirect inbound connections from each user to his specific server instance. By doing so, multiple users can access the CDSS server over the same protocol on the standard port without requiring any special configuration by the user. Each user can only communicate with the server instance that was started explicitly for him and that has been automatically configured by CDSS to allow access only to the files that he has requested.

CDSS is currently implemented as a collection of web and shell scripts that run on Linux servers that support the IPTables and IPChains firewalling systems. CDSS currently supports image distribution via the following protocols: HTTP, FTP, TFTP, NFS, SMB, and AppleShare IP. CDSS can share any filesystem image file stored on the server as well as the individual contents of those images that the server can loopback-mount.

Introduction

In the last five to ten years, many systems have been written to assist in automating the installation and management of large collections of homogeneous servers administered by a small team of systems administrators. These programs have simplified this complex administration task substantially. However, not all installations enjoy the efficiencies provided by such an infrastructure. Many sites have to deal with large heterogeneous infrastructures where there are many system administrators each controlling a small cluster of machines. In this environment, distributing software in a secure, reliable, efficient and effective manner can be quite difficult. In this paper we present the CD Sharing System (CDSS), a collection of scripts and web interfaces for enabling and simplifying the distribution of software and installation images to heterogeneous clients. First we will discuss common distribution solutions before presenting the server-per-user IP redirection solution used in CDSS. We will also cover the specifics of implementing CDSS over common file distribution protocols provided by UNIX servers. Finally, we will discuss the use of the server-per-user IP redirection technique in other applications and future enhancements planned for CDSS.

A very critical element of a software distribution system is security, particularly with proprietary software under a site or multiple-use license. Software vendors rarely condone the wide-scale distribution of images to non-licensed parties, so the goal of a software distributor is to make the software easily accessible to the licensed parties while denying access to everyone else. When the number of licensed parties is very small (a single group) or very large (freely-distributable software), this is not difficult. In the non-trivial cases, there are a few possible solutions.

One solution is to manually distribute software installation media to each eligible group. Each group can then individually convert the image into a format that is best for them. However, this method does not scale to large numbers of groups, and causes significant duplication of effort both on the part of the distributor and the users.

Another possibility is to place the images and image contents on a central distribution server and provide access to the images over standard protocols. A system like this would necessarily require a substantial cross-protocol authentication and access control system to prevent unauthorized access. Unfortunately some of the most important protocols have no authentication systems at all, and the others that do often do not share a common authentication mechanism. This eliminates some protocols from central administration entirely (e.g., TFTP), and makes managing the other protocols a significant administrative hassle.

CDSS is a framework for software installation media image distribution that provides simplified maintenance, enhanced security, and substantial platform independence. CDSS achieves this by running unique and isolated versions of the sharing software for each and every requesting user. On a server

providing images to six users, there may be two web servers, three NFS servers, and six FTP servers all running simultaneously. Even in a situation like this the existence of multiple server instances is transparent to the users, as they are automatically forwarded to their unique server instance using the transparent IP redirection provided by the Linux IPTables and IPChains firewall systems. By running a separate server instance for every user, we can disable the built-in authentication systems of each protocol entirely, simplify the configuration and centralize the authentication within CDSS, all without requiring any special configuration on the client's software. Finally, users request and manage their images from a simple web interface that gathers the information necessary to create the share and runs the sharing script.

The CDSS Solution

At its core, CDSS performs five major tasks:

- Image Selection: Users visit a web page listing all of the available images. They select the specific images they need access to, supply the necessary passwords for those images, and specify the IP addresses that will have access to the images.
- Image Preparation: The images that the user has requested are linked into a directory created for the user, and loop-back mounted into this directory.
- Server Configuration and Execution: The servers necessary to provide the protocols requested by the user are configured to allow access to the user's directory and started.
- Firewall Configuration: Once the servers for each protocol have been started, rules are inserted into the running firewall configuration to redirect packets from the user to their non-standard port for each protocol.
- Security: The scripts also ensure that individual images and directories of images are not shared to to those who do not know the required passwords for those images.

Image Selection

The first step in any sharing session is for the user to make an image request. Each request needs to contain a set of images, any passwords required for those images and the IP addresses that will access the images. In order to simplify this process, CDSS contains a relatively simple web interface that allows the user to easily select the set of images they want to share and the protocols that they need access over. The web interface builds the list of available images at runtime, so there is no special configuration required to share a new image other than to place it into the image repository. The images can also be password protected on an individual or per-directory basis. It is also possible for the server administrator to share a set of images directly, without using the web interface.

Figure 1 shows the client web interface on a small CDSS server. We can see that there are multiple directories that contain images. Each directory and file can have a password or set of passwords associated with it. If a user wants to access several files that require passwords, they can place the passwords in the input box for each directory, or they can place multiple passwords in a single input box. We also request a separate username and password from the user at request time. This username and password is used to identify the user's share and to prevent unauthorized removal of active shares. Along with images and passwords, users must select a set of protocols and specify any additional IP addresses that also need access to the images.

Image Preparation

Once a user has successfully requested a set of images, they must be prepared for his use. Initially, all the images are stored together in a master directory

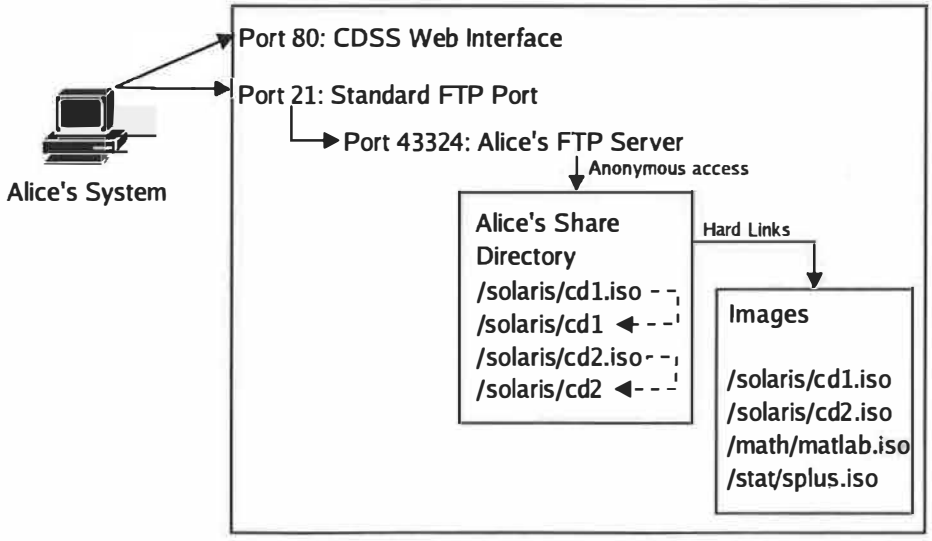


Figure 1: A CDSS server with one client.

tree. When a user makes a request, a new directory is created on the same partition as the master tree and the images requested are hard-linked into the newly-created directory. At this time, we also create individual subdirectories for each image and loop-back mount the images into these directories. This allows the requester to access the contents of each image directly, without having to download the image in its entirety. The loop-back mount can even be used for complete network-based installations directly, without the need to re-share the images. Because the images are loop-back mounted, any CD image that can be mounted by the system can have its contents shared by CDSS. Non-mountable images such as audio CDs can be shared, but only as raw images (the individual tracks or files can't be accessed).

Server Configuration and Execution

After the individual user directory has been created, the servers that provide the requested protocols need to be configured and started. It is important to note that multiple server instances are started for each user, one per protocol. Each server instance is configured to listen for in-bound connections on a non-standard port that is specific to that user. This port is chosen randomly and has no relation to the standard port that the protocol runs on. For each protocol we have a single standard configuration file, if one is required. Before the server instance is started, CDSS copies the standard configuration file into the user's image directory and makes any necessary changes to indicate the directory to be shared and the port to listen on. Then the server is started. Running a separate server instance for each user allows simple automated configuration and enhanced security. However, the redundancy in this system does place an increased load on the server due to the large number of active server instances. CDSS uses lightweight server programs when available and configures them with a minimum of options in order to reduce the CPU and memory usage of the system as a whole.

Firewall Configuration

Once the system has prepared the images and started the necessary servers, the firewall must be configured to allow connections from the requesting user. Connections to the standard ports for each protocol must also be correctly redirected to the unique port allocated to the user for that protocol. Finally, the firewall rules must prevent non-authorized users from connecting directly to the server instances, bypassing the redirect in place on the standard port for each protocol. CDSS currently supports the Linux IPTables and IPChains firewalling systems, although it could be extended to support any firewalling system that can accept rule changes in real-time and transparently redirect connections to alternate ports on the same machine.

Security

Although the firewall configuration described above provides a large measure of security against the possibility of unauthorized access to files, it is also important to secure the web interface effectively. While the web interface is a very convenient way to provide access to large numbers of users without administrator intervention, it also creates the risk of exposing a setuid-root program to the entire allowed user base. CDSS uses extensive checking and verifying of user-provided input both in the web interface and in the master sharing script to prevent the compromise of the hosting server. The CGI web interface itself runs as an unprivileged user and executes the setuid-root script after the user has finalized the request in the web interface. Although it would be ideal to not have to run any part of the system as root at all, root-level permissions are required to maintain the loop-back mounts and adjust the firewall rules. The master sharing script is exceedingly cautious about verifying the data provided by the user and the programs that it executes as root.

For most applications, the CDSS system will need to run on a server that has no unprivileged accounts on it as the media images are stored as world-readable files. It may be possible to use filesystem permissions on the master tree and the individual user directories. However, CDSS does not currently support this functionality. The substantial majority of images shared by CDSS are ISO 9660 images or similar. Images such as these have no access controls inherent in them. However, it is possible to share images that do have filesystem access controls. If this is the case, only those files that are world-readable can be accessed remotely. The loop-back mount system has no ability to edit a mounted image, so we must be careful to avoid images that might be restricted in this way. Such restrictions are also largely ineffective, as the user can always download the complete image and bypass the restrictions at their leisure.

Beyond the security risks inherent in running the CDSS software itself, there is also the possibility of vulnerabilities in the server software for each protocol. In writing CDSS, we have attempted to choose packages with excellent security histories and only the features we need. However, we do extensively rely on the ability of the server software to restrict the users to their assigned directories. Server programs that are unable to restrict access to a single directory tree cannot be used with CDSS. If flaws in the software allow users to break out of their individual directories, then any image file on the system could be accessed and downloaded. In situations where this is a significant concern, additional security could be introduced to the system by chrooting or jailing the server processes into the individual user directories.

Features

Along with the primary tasks listed above, CDSS also has several features that enhance the usefulness of the system, even though they are not required for minimum functionality.

Web Interface

CDSS contains a simple and powerful web interface that allows for individual users to arrange for any number of images to be shared to them with no administrator intervention. The web interface allows for the removal of mounts in order to allow a user to make multiple requests in a single day.

Universal Access

The CDSS system provides enhanced security and efficiency when distributing software installation media images across a large organization. When a user requests an image, they are allowed to specify the IP addresses that are able to access this images being shared. The means that you can share images to machines that are half-way around the world, at home, or even not installed yet. This flexibility is important in larger organizations, where it might be necessary to support remote users without incurring the costs and risks involved in sending CDs to the remote sites, where security and staffing may be a problem. It is also possible to restrict access to the system to IP addresses within the organization to prevent sharing to unauthorized systems.

Securing Any Protocol

CDSS also enhances the security of some protocols that were originally designed with no security mechanisms at all. The most common of these is TFTP, which is primarily used to distribute firmware images to network devices and provide boot-up files to disk-less machines and automated installation systems. Other than blocking access to TFTP entirely, it is impossible for a vanilla TFTP server to provide different sets of files to different clients. By running individual servers for each client, CDSS allows one master server to distribute files over TFTP to many clients without requiring specific or difficult setup on the TFTP client. Other protocols like HTTP and NFS also benefit from this feature, although they do have rarely used authentication systems of their own.

Implementation Details

Although the operational concepts behind CDSS are relatively simple, there are some subtleties in implementing the firewalling and sharing each of the protocols supported by CDSS. We will go over the basics of the program, and then discuss the methods and server software that we chose for each protocol.

Core Operation

The central functionality in CDSS is implemented in a setuid-root perl script called share.pl. Share.pl takes a number of arguments, including the username of the user sharing the files, the IP addresses to be granted access, the files being requested and the passwords for the images. share.pl runs under the perl taint system, so all of the user-provided data must be checked to ensure that it does not contain any malicious strings or data. Once the data has been checked, a directory is created for the user, and the images requested are hard-linked into that directory. Hard-linking the files allows the servers to chroot themselves into the individual user directory without actually copying the files. Hard-links also eliminate any possible problems with access permissions or the inability of sharing software to follow soft-links. However, hard-linking requires that all of the shared files must reside on a single disk partition. CDSS servers commonly use RAID or LVM in order to get drives that are large enough to hold the substantial amounts of data on installation media.

As a convenience to the user, we also loop-back mount the requested images into newly created subdirectories for each image. Although loop-back mounting of images is fully supported by the Linux kernel, it is not enabled in some distributor's default kernels. Even in the kernels where it is supported, the maximum number of simultaneous loop-back mounts is limited to 8. Adjusting this requires modifying the max_loop constant in the drivers/block/loop.c file in the kernel source. Because of this limitation, every CDSS server will need a custom compiled Linux kernel.

The configuration files that are used to start the server instances are not the files that initially come with the software. CDSS has a special repository of custom configuration files that are designed to be automatically configured for the per-user alternate port system used by CDSS. In general, the files themselves undergo few changes for use with CDSS. Once the configuration files are in place, the servers are started and the firewall rules are configured to allow access.

When a user has completed using a share, they are expected to return to the web interface and remove it. However, it's very common for users to neglect to do so. CDSS comes with a shell script intended to be run out of cron that will automatically remove any share that has been in place for more than 24 hours. Shares that are needed for more than 24 hours can be specially configured, but are discouraged. It could be possible to use the automount system to manage the loop-back mounts used by the CDSS system, but the automount systems we looked at aren't well suited to CDSS. The automounter interfaces directly with the kernel, and only a single automount process should be run on a system at a time. Also, most automount systems do not have the ability to make other arbitrary changes to the system when it is time to unmount an active drive, and for CDSS, the unmounting the loop-back is only one of the steps in removing a share.

Firewall Operation

CDSS relies heavily on the IP redirection features that are part of the Linux IPTables and IPChains

host-based firewalling systems. Without IP redirection, every user would have to specifically configure their clients to connect with the server on alternate ports. Some lightweight clients for network hardware devices and embedded systems do not have systems to allow for this level of customization. The IP redirection allows the system to support multiple users on the same port simultaneously without requiring any special configuration on the part of the client.

The firewall is configured as follows. A unique chain or table is created for each requesting user, and all of his rules are placed into this chain. With the IPTables system two chains have to be created, one in the PREROUTING table, and one in the INPUT table. We then link these newly created tables into the primary tables. All of the rules for a particular user are placed into these tables, so that they can be easily removed. Most protocols only require two firewall rules: one to allow traffic from their IP to the randomly allocated port for this protocol, and one redirecting the user's traffic from the standard port to the randomly allocated one. However, some protocols (such as NFS) require a more involved setup.

Here are the commands run by CDSS to setup to firewall for a HTTP-based share under the IPChains firewall system. The IPTables firewall requires a few more commands because the redirect target is only supported in the nat tables. Figure 2 shows two clients accessing shares and how the IP redirection allows them to simultaneously access two different server instances on the same port.

In this example, most variables are self-explanatory. The actual server for this user resides on $redirport, and $ip is the IP of the client for this image.

```
$IPCHAINS -N $username
$IPCHAINS -I $SYSTEM_CHAIN \
   -i $INTERFACE -j $username
$IPCHAINS -I $username \
   -p $proto -s $ip -d $MYIP \
   $dport -j REDIRECT $redirport
$IPCHAINS -I $username \
   -p $proto -s $ip -d $MYIP \
   $dports -j ACCEPT
```

When it is time to remove a share, the removal script flushes and removes the entire chain created for that user and the rule in the system chain that activates the user's chain.

Protocol Specifics

Currently, CDSS supports file access over the following protocols: HTTP, FTP, TFTP, NFS, SMB, and AppleShare IP. In this section, we will look at each protocol and discuss the server software used and any non-standard configuration that is necessary.

HTTP

As a protocol with near-universal support, HTTP can provide quick, convenient access to almost any client. We chose Boa [5] as the server for the HTTP protocol as it is very lightweight, easy to configure, and high-performance. The only configuration options that need to be set are the port and the directory to be shared, and no special firewalling rules are necessary. Since the client web interface runs on the standard web port, we provide HTTP access on port 8080, the standard secondary port for HTTP. Since this is a non-standard port, we include a clickable link in the output of the sharing script. Since execution over HTTP is not commonly supported, HTTP is most often used to get a single file or files out of a large image without having to download the complete image. Boa also provides internal support for HTTP continue and internal index generation.

TFTP

TFTP is a key protocol for CDSS to support, both due to its extensive use in automated installation and embedded systems, and because it lacks any authentication system at all. By providing TFTP through the CDSS system, a layer of security can be placed in front of this normally insecure program. In many environments, the security issues of TFTP are

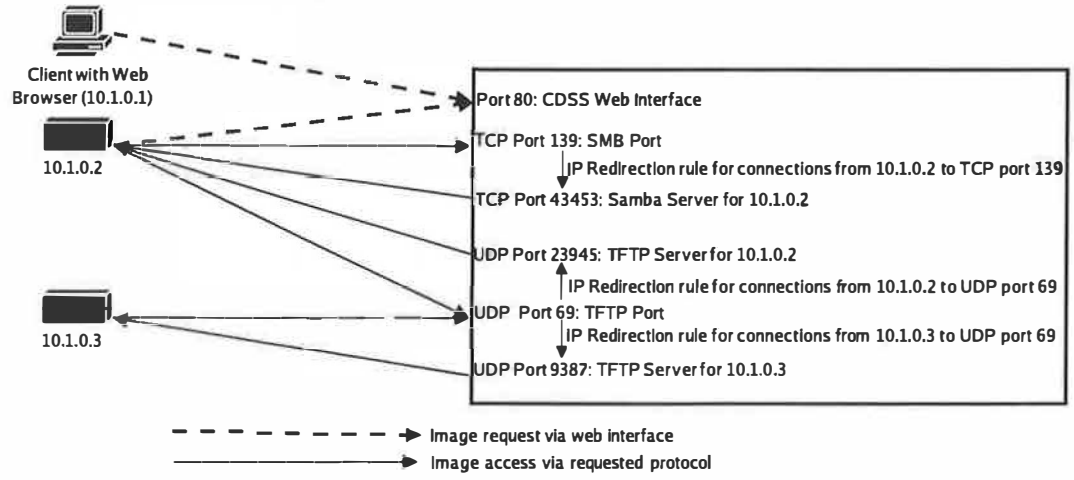


Figure 2: Multiple clients access a CDSS system.

ameliorated by firewalling the TFTP port to the small set of IPs that the TFTP clients reside on. CDSS allows for a central distributor to maintain a TFTP server without the overhead of maintaining a lengthy firewall access list.

TFTP is traditionally run out of inetd, running on a UDP port. In a CDSS environment, we use netcat compiled with the -DGAPING_SECURITY_HOLE. That allows the use of the -e argument, which emulates inetd for a single program. Because netcat only allows a program name to be provided with the -e argument, we also create a simple shell script that contains the arguments to tftpd. Other than the script and the standard firewall rules, no special configuration is needed for tftp.

AppleShare IP

AppleShare IP is a recent addition to the Apple file sharing protocol. Before its release in MacOS 8, AppleShare was only available over the AppleTalk protocol. However, we can now use AppleShare IP to provide access to the images on the CDSS system to older Macintosh systems that are not running MacOS X. (MacOS X supports Windows sharing as well.) For MacOS systems prior to MacOS 8, the only supported option is to copy the files over using HTTP or FTP. We use the netatalk package for the actual sharing, and no special firewall rules are required beyond the single redirect.

SMB

File sharing for Windows systems is provided by the SMB protocol. Although it is quite a complex protocol, we only need to provide the sharing portion of it. The most difficult part of sharing over SMB is actually client support rather than server support. The SAMBA package has excellent support for SMB and is easy to configure, but many windows machines have difficulty accessing IP-based shares. However, with some careful configuration, most windows machines can access IP-based SMB shares.

Configuring Windows clients to access IP-based SMB shares can be quite difficult. Traditionally, Windows has used an Ethernet broadcast protocol known as NetBIOS to enable file sharing between windows machines. When IP based sharing was introduced in windows NT, the WINS system was created to create linkages between NetBIOS names and IP addresses. Unfortunately, the WINS and NetBIOS systems often cause difficulties in accessing IP shares. We have had the most success with Windows NT, Windows 2000, and Windows XP machines, although Windows 95, Windows 98, and Windows ME machines can attach to IP-based shares if NetBIOS is completely disabled and the hosts are correctly listed in DNS or the Windows LMHOSTS file, which stores linkages between machine names and IP addresses.

NFS

Of all the protocols that we support, NFS requires the most complex configuration. On most systems, NFS support is provided by the kernel itself. However, since CDSS requires multiple running copies of the same server, we have to use the user space NFS server. The user space NFS server consists of multiple small programs that each support one part of the NFS protocol. CDSS requires only nfsd and mountd. An interesting element of NFS is that is traditionally runs on randomly allocated ports, unlike all of the other protocols listed here. In order to direct clients to the proper port, the nfs daemon registers itself with the RPC portmapper. When clients want to mount an nfs share, they discover the actual port for NFS from the portmapper, and then connect directly.

Because we are using IP redirection to move the connections to a different port for every user, we have to override the redirections made automatically by the portmapper. This is performed by replacing the portmap configuration after each NFS server is run. By replacing the configuration with a standard one, we can then configure our IP redirection system to correctly send each client to their individual NFS servers. This sort of configuration replacement is necessary for any protocol that uses the RPC system to manage its communication. Unfortunately, some systems do not have a user space NFS server, or they do not allow for the NFS server to open a specific port. Non-standard NFS systems or modifications to the Linux user space NFS server may be required in these cases. Because of the transient nature of CDSS shares, we strongly recommend that clients using NFS configure their mounts as soft mounts. This allows the client to kill processes waiting for data from the NFS server or unmount the share entirely if their share has been removed.

Other Applications

Although CDSS was written to share software installation media images, it could easily be extended to provide access to programs or other services. It's important to note that we have endeavored to use lightweight servers and protocols whenever possible. Although it is possible to use the IP redirection techniques of CDSS with more substantial systems, the ability of the system to service multiple users would be hindered. CDSS is best suited to protocols without authentication built-in, or when different users need completely different configuration and the software does not support having per-user settings.

Conclusion

This paper introduces a system for distributing software installation media images to a large set of heterogeneous clients. By using a unique server instance for every user, we are able to simplify and largely automate the configuration of the unique server for each user. Even though we may be running many copies of the same server, per-user IP redirection allows all of the server instances to appear on the standard port for the system, even though they are

actually completely separate. This system allows for enhanced security and efficiency while reducing the administrative overhead of maintaining such a server.

Availability

CDSS is licensed under the GPL and is currently being hosted at SourceForge. The master web site is http://cdss.sf.net/ .

Author Information

Ted Cabeen has been working with UNIX systems for 10 years, and has been administrating them professionally for the last six. He developed CDSS at the University of Chicago and now works for Impulse Internet Services in Santa Barbara, CA. He can be reached at secabeen@pobox.com .

Job Bogan is currently working as a consultant specializing in clustering and administration for large academic institutions and companies.

References

[1] SMCC, "LOFS: Loopback Virtual File System," *SunOS 5.5.1 Reference Manual*, Section 7, Sun Microsystems, 20 March, 1992.

[2] *Linux IP Firewalling Chains*, http://www.netfilter.org/ipchains/ .

[3] *The Netfilter/IPtables Project*, http://www.netfilter.org/ .

[4] Olaf Kirch, *Universal NFS Server for Linux*, ftp://linux.mathematik.tu-darmstadt.de/pub/linux/people/okir/ .

[5] *The Boa Webserver*, http://www.boa.org .

[6] *Netatalk, A Kernel Level Implementation of the AppleTalk Protocol Suite*, http://netatalk.sf.net/ .

[7] *Samba, A SMB/CIFS Server Suite*, http://www.samba.org/ .

[8] *ProFTPD, Highly Configurable GPL-Licensed FTP server Softwarek*, http://www.proftpd.org/ .

Virtual Appliances for Deploying and Maintaining Software

Constantine Sapuntzakis, David Brumley, Ramesh Chandra, Nickolai Zeldovich, Jim Chow, Monica S. Lam, and Mendel Rosenblum – Stanford University

ABSTRACT

This paper attempts to address the complexity of system administration by making the labor of applying software updates independent of the number of computers on which the software is run. Complete networks of machines are packaged up as data; we refer to them as *virtual appliances*. The publisher of an appliance controls the software installed on the appliance, from the operating system to the applications, and is responsible for keeping the appliance up to date. These appliances can be configured by users to fit their needs; the configuration is captured such that it can be reapplied automatically when the appliance's software is updated. We have developed a compute utility, called the Collective, which assigns virtual appliances to hardware dynamically and automatically. By keeping software up to date, our approach prevents security break-ins due to fixed vulnerabilities.

This paper presents the concept of virtual networks of virtual appliances and describes our prototype of the Collective Utility. We demonstrate the feasibility of our approach by creating appliances for groupware servers, Windows desktop environments, and software development environments.

Introduction

On July 24, 2002, Microsoft released a patch for buffer overruns in SQL Server 2000 [11]. Six months later, on January 25, 2003, the SQL slammer worm inundated network links with packets, slowing Internet connections and costing an estimated $1 billion. The worm exploited a vulnerability on unpatched servers [19]. Unpatched software affects more than just services; desktop systems are also in jeopardy when security patches go unapplied. On June 5, 2003, Stanford University disabled all outgoing mail delivery due to the BugBear.B virus, which was leaking confidential documents [20]. The hole exploited by BugBear.B was fixed by Microsoft in a patch [10] issued more than two years before, but many users had not updated their desktops.

These two incidents underscore the importance of keeping systems up to date with respect to security patches. But security patches are released frequently, and end users may not be aware of patches or have the know-how to update their systems. Patching today is done through a variety of ad-hoc mechanisms; applying a patch sometimes breaks a system. To improve security, we must make updates automatic, reliable, and even mandatory.

Software update is only one of the problems facing system administrators. Setting up and maintaining a computing infrastructure requires much effort. While large organizations may have IT departments, smaller organizations, such as start-up companies and university research groups, may not have professional staff to create and manage infrastructure. With home users, the situation is even worse. They are often poorly versed in system administration and waste much time as a result.

Approach

We observe that computers do not have to be difficult to install and maintain. The TiVo personal video recorder has much of the same hardware and software as a PC, yet it automatically downloads updates and installs them, without hassling the user. Computing appliances, like the TiVo, provide a more predictable environment for software updates since users do not install software. Instead, the software installed on the appliance is controlled by the appliance vendor, who can test all the software to ensure it works together before distributing it.

Inspired by the ease of administering of appliances, we have proposed organizing software systems as *virtual appliances* in previous work [17]. A virtual appliance (VAP) is like a physical appliance but without the hardware; as such, a VAP is like software and can be shipped and stored electronically. Like the software in a real appliance, the software in the virtual appliance is written to run on top of hardware. We chose x86 hardware due to the vast amount of x86 software and hardware available. Rather than run the VAPs on bare hardware, we run them on an x86 virtual machine monitor, VMware GSX Server, to ease management. Recognizing that network management now plays a substantial part in system administration, we have extended the concept of a virtual appliance to include the network. A virtual appliance can be a network of virtual appliances, which we call a *virtual appliance network* (VAN). For example, a groupware VAP may consist of separate DNS, LDAP, web, and mail VAPs, connected by a virtual network, and protected by a firewall VAP. By bundling appliances into VANs, we amortize the cost of network administration among users of the bundle.

Appliance publishers create, publish, and update VAPs; like software publishers, they will often be organizations but may just be sophisticated individuals. Users get copies of VAPs from publishers and run them. Instead of installing software in VAPs, users acquire the features they need by getting additional VAPs.

We propose running VAPs on a compute utility. The utility automatically manages hardware, deciding which appliances run where. Appliances are not tied to specific hardware and can be moved to balance load or route around failures. Professionals, called *utility administrators*, procure and maintain the utility's hardware; they also install and maintain the utility's software.

An appliance's software is stored on virtual disks provided by the publisher; we call these disks *program disks*. The publisher controls the contents of program disks and can publish new versions to update the appliance's software. When an appliance is restarted, the utility will automatically pick up the most recent versions of the program disks unless instructed otherwise. To allow customizations and data to persist across updates, data is stored on separate *data disks* or in network storage.

By automating software update, our proposal makes software administration independent of the number of computers at a site. Not only does this reduce cost, but making software easy to update improves security, and reducing the management overhead encourages more software to be used.

Overview

This paper presents a prototype system, called the Collective, designed to support the creation, publication, execution, and update of VAPs. We also report our preliminary experiences with the system.

The Collective has three main components: a configuration language called CVL for describing VAPs, an appliance repository for publishing VAPs, and a utility for running VAPs. The Collective Virtual appliance Language (CVL), pronounced "civil," describes a VAP, including its parameters and, for VANs, the network layout. Appliance repositories allow publishers to post VAPs and to update them. Users refer to a repository using a URL when telling the Collective what appliance they wish to run. Finally, the Collective Utility manages a cluster of computers, runs VAPs, and performs updates.

The paper discusses the following in more detail:

- **Specification and implementation of VANs.** Just as virtual machines make whole computer states readily manipulable, virtual appliance networks ease the manipulation of networks of computers. We have implemented techniques for starting, stopping, and updating VANs. Using CVL, we describe how to compose appliances into VANs and how to attach VANs to other VANs.
- **Configurable and extensible VAPs.** To be reused, published appliances must be customizable to fit the needs of the user. Using parameters set in CVL files, users can configure VAPs with such details as network parameters and domain names; these parameters are passed by the utility to a VAP on boot and on update. Also, in the CVL language, a derived appliance inherits parameters from its parent and can thus automatically take advantage of changes made to the parent appliance. Still, users can override parameters from the parent in the derived appliance to customize it for the local site.
- **Update support.** The Collective helps publishers maintain users' software installations by providing a predictable update model – replacing the program disks of the appliance. Since users refer to appliances from repositories, the Collective Utility can directly consult the repository to find the most up to date versions and even prevent users from running vulnerable software if that is desired. Also, the Collective Utility can minimize downtime when updating a running VAN by restarting only the modified appliances.
- **Experiences.** This paper also reports our preliminary experience with the system in three scenarios: (1) a network of common group services such as DNS, LDAP and Mail, (2) Linux software development environments, and (3) Windows desktops. Our experience suggests that VAPs are a feasible way of maintaining software.

Paper Organization

The rest of the paper is organized as follows. The next section motivates our design with some examples. Then, we explain how we specify a virtual appliance network. Subsequently, we present the design of the appliance repository and an overview of the interface of the Collective Utility and its implementation. After that, we describe our experiences and related work. Finally, we present our closing remarks and conclusions.

Motivating Examples

Virtual appliances reduce system administration by having one organization, the appliance publisher, manage the software for all users of an appliance. Because the publisher controls the operating system, shared libraries, and applications in the appliance, the publisher can test them and ensure they work together.

Networks of virtual appliances have performance, isolation, and maintenance benefits over individual appliances. The performance benefits come from being able to run multiple appliances on multiple computers in parallel. The isolation benefits come from separating services: for example, firewall, LDAP, DNS, and mail servers can each have their own appliance. Updating and rebooting an appliance affects a single service or, if the service is replicated, a fraction thereof.

The maintenance benefits come from two sources. First, the cost borne by the publisher when maintaining an appliance is amortized over all users of the same appliance. Second, since a virtual appliance network can specify topology and network infrastructure services like DHCP and DNS, the user does not need to deal with the complexity of setting up networks.

We illustrate the use of virtual appliances with three concrete scenarios:

- **Groupware.** Many organizations need groupware tools to collaborate. Groups often find it difficult to create and keep their groupware up to date, especially in non-technical organizations. As a result, many departments that would benefit from their own groupware go without for lack of administrators.

  Virtual appliance networks make it possible to bundle together a number of common services, such as DNS, LDAP, and Mail, and release them as a unit. In keeping with security best practices [12], which suggest that each service run on a separate operating system, these functions are split across multiple appliances connected by a virtual network. With the Collective system, we can instantiate a new network of groupware appliances quickly and keep them up to date.
- **Software development.** Some software systems are difficult to compile, link, and install. They have been well tested only on specific versions of tools and are picky about where libraries and files are placed. It is time-consuming to track down and install the tools necessary for a complete build.

  Instead of creating an installer, the software publisher can bundle the necessary tools in an appliance and distribute it to users. Each user runs a copy of the appliance and uses the tools. To share data between the appliance and other systems, the user mounts a network file system into the appliance.
- **Telecommuters' desktops.** The advantages of appliance and utility computing can also be applied to desktop computing. In this scenario, the IT department gives a user a configured network of appliances that the user can run at home. For example, a standard office worker may have a productivity appliance with an office suite and e-mail tools. To allow the user to access company resources from home, the company bundles a VPN appliance with the productivity appliance. The VPN appliance also has a firewall to protect the productivity appliance and the company data on it from being attacked by other appliances.

Virtual Appliance Networks

We use the term *virtual appliance* (VAP) to refer to either a virtual machine (VM) appliance or a virtual appliance network (VAN) composed of VAPs. To support customization, VAPs are configurable; the behavior of a virtual appliance can be changed by changing the values of parameters.

The interface between a VAP and the Collective Utility takes the form of a *pre-defined set of parameters*. These parameters are either used by, or set by, the Utility. For example, for each VM appliance, the Collective needs to know the name of the VMware `.vmx` configuration file. For each VAN, the Collective needs to know the network topology, the appliances connected to the network, and the dependencies between them. The Collective can also assign values to parameters. After allocating certain resources, such as IP addresses, to an appliance, the utility sets parameters on the appliance corresponding to these resources; the appliance and other appliances can use these parameters to configure themselves.

Every VAP can also have *appliance-specific parameters* that are specific to configuring the software in the appliance. While complex software packages such as OpenLDAP have tens of configuration parameters, the appliance publisher can reduce this number by providing sensible defaults, thus saving the users the time-consuming task of learning all the configuration parameters in a package. Furthermore, the publisher of a VAN can pre-configure the appliances in it to talk to each other or propagate parameters between them, freeing the user from having to manually create these connections or propagate values.

An appliance specification includes a set of parameters and values. Parameters can be set by appliance publishers, the user, and the Collective Utility. A publisher may wish to compose a network of appliances out of other published appliances. Or, he may extend an existing appliance by assigning specific values to some of the parameters and republish it. For example, a university system administrator may inject the university domain name into an appliance and make it available to all users. A user may further extend the appliance by, for example, supplying the appliance with credentials in order to gain access to their data. Updates should propagate down a chain of publishers while maintaining customization; if the original publisher updates the appliance, the extended ones should update automatically, maintaining customizations wherever possible.

It is thus desirable that our configuration language support composition, extension, and allow changes to a VAP to be propagated to extended versions. This argues for a configuration language that supports abstraction and inheritance. To satisfy these requirements, we have defined the Collective Virtual appliance Language (CVL).

The CVL Language

The CVL language, version 0.8, has a generic syntax suitable for describing configurations of any types of objects and a set of pre-defined objects that model the semantics of virtual appliances.

An object may consist of component objects, a set of parameters, and possibly their values. An object can inherit from one other object. The value of a parameter is set using an assignment statement. Assignments in parent objects are executed before assignment in the derived objects, allowing specialized assignments to override the generic. Without looping constructs or even conditional statements, the language is far from being Turing-complete. It is a simple configuration language whose goal is to generate parameter and value pairs for each object. For reference, the BNF grammar for the CVL language is shown in Appendix A.

The semantics of virtual appliances are captured by four pre-defined types of objects:

- **Interface** objects represent virtual Ethernet network interfaces in VAPs.
- **Appliance** is the base object for all appliances.
- **VMAppliance**, inheriting from Appliance, is the base object for VM appliances. VMAppliance has a vm parameter that points to the contents of the virtual machine, which is a .vmx file in the case of a VMware virtual machine.
- **VANAppliance**, also inheriting from Appliance, is the base object for VAN appliances.

```
Interface {
  var "required" mac, ip, subnet, netmask;
  var defaultroute;
}

Appliance {
  var requires, provides;
  var "required" vanIF;
}

VMAppliance extends Appliance {
  var "required" vm;
  var datadisks;
  Interface ethernet0;
  vanIF = "ethernet0";
}

VANAppliance extends Appliance {
  var defaultroute;
}
```

Figure 1: Pre-defined objects in CVL.

Figure 1 shows all of the parameters defined for each of the base objects. The semantics of these parameters are discussed in more detail below. The pre-defined objects and their parameters are used by the Collective Utility in configuring and running virtual appliances. We can set the values of these parameters and add new appliance-specific parameters by deriving new appliances from either VMAppliance or VANAppliance. Only VANAppliances can have VMAppliance components; only VMAppliances can have Interface components. Currently, CVL does not allow the definition of any other kinds of objects.

From http://virtualappliance.org/DNS:

```
/* Language version number */
CVL = "0.8";
DNS extends VMAppliance {
  var "required" domain, dnshosts;
  var port;
  port = "53/udp";

  /* Virtual machine configuration */
  vm = "dns.vmx";
  datadisks = { device => "ide0:1",
                size   => "100mb" };
  /* Dependencies between appliances */
  provides = "DNS";
}
```

From http://virtualappliance.org/OpenLDAP:

```
CVL = "0.8";
OpenLDAP extends VMAppliance {
  var port, sport;
  port = "389/tcp";
  sport = "636/tcp";

  /* Virtual machine configuration */
  vm = "ldap.vmx";
  datadisks = { device => "ide0:1",
                size   => "100mb" };

  /* Dependencies between appliances */
  provides = "LDAP";
  requires = "DNS";
}
```

From http://virtualappliance.org/Firewall:

```
CVL = "0.8";
Firewall extends VMAppliance {
  Interface ethernet1;
  var services;

  /* Virtual machine configuration */
  vm = "fw.vmx";
}
```

Figure 2: A few virtual appliances used in Groupware.

Figures 2 and 3 show an example of a groupware network called Groupware with three components, a DNS server, an LDAP server, and a firewall. We will use this example in the rest of the section to explain the CVL language. We first describe how to specify components and how to declare parameters and assign to them, and then describe the semantics of the parameters in the pre-defined base types.

Components

Component VAPs of a VAN must be *imported* into the name space of the file defining the VAN. The import statement specifies the appliance definition to be used and a short name by which the definition is referred to. The definition includes the URL of the appliance repository, the name of the .cvl file, and optionally a version number. If no version number is specified, the latest version is assumed.

Specifying a particular version of an appliance is useful in cases when only that specific appliance version supports some feature. Additionally, a specific version of an appliance may be desired when a set of appliances of certain versions have been tested to work together, as is the case of the Groupware appliance. Most of the time we expect the user not to specify a particular appliance version when starting an appliance, thus allowing the

appliance software to be automatically updated as new versions become available.

From http://virtualappliance.org/Groupware:

```
/* Language version number */
CVL = "0.8";
/* Import component appliance definitions */
import {
  url => "http://virtualappliance.org/DNS",
  cvl => "DNS.cvl",
  version => "3"
} DNS;

import {
  url => "http://virtualappliance.org/Firewall",
  cvl => "Firewall.cvl",
  version => "5"
} Firewall;

import {
  url => "http://virtualappliance.org/OpenLDAP",
  cvl => "OpenLDAP.cvl"
} OpenLDAP;

Groupware extends VANAppliance {
  var "required" domain;

  /* components */
  DNS d;
  OpenLDAP l;
  Firewall f;

  /* configuration */
  d.domain = domain;
  l.domain = domain;

  d.dnshosts = { name => l.name,
                 ip   => l.ethernet0.ip },
               { name => d.name,
                 ip   => d.ethernet0.ip },
               { name => f.name,
                 ip   => f.ethernet0.ip };
  f.services = { port => l.port,
                 ip   => l.ethernet0.ip },
               { port => l.sport,
                 ip   => l.ethernet0.ip },
               { port => d.port,
                 ip   => d.ethernet0.ip };

  /* network topology */
  vanIF = "f.ethernet1";
  defaultroute = f.ethernet0.ip;
}
```

Figure 3: A virtual appliance network for running Groupware.

Components in VANs are declared by specifying the appliance type followed by the name of the component. For example, the Groupware appliance in Figure 3 declares that it has three components: a DNS server named d, an OpenLDAP server named l, and a firewall named f. The import statement for the DNS appliance specifically requests version 3 of the appliance from the repository, while the import statement for the Firewall appliance does not specify the version number, so the latest version present in the repository will be used.

Parameters and Assignments

Each appliance inherits all the parameters from its parent appliance and can define new parameters. It may assign to its parameters or the parameters defined in any of its components. We refer to parameter v of an object o as o.v.

Parameters may be given attributes. Parameters declared with the attribute "required" must have a value before the appliance can be started. With this construct, the Collective Utility can detect errors early and help users by providing them with a meaningful error report. As shown in Figure 2, the OpenLDAP appliance has a required domain parameter, which requires the user to set the domain name for the LDAP server. The Firewall appliance, on the other hand, has an optional services parameter, which allows the user to specify what services the firewall should expose. If the value of the services parameter is not specified, the firewall appliance will still function without exposing any services.

Some parameters, like user keys and passwords, are sensitive; we declare such parameters with the attribute "sensitive". Sensitive parameter values should not be stored unencrypted in the file system. Instead, before sending parameters to an appliance, the Collective Utility passes them to a user agent service, which keeps a cache of sensitive values. If the requested parameter is not present in the cache, the agent prompts the user for the value. To allow sensitive data to persist across instances of the user agent, the user agent stores the data in a file encrypted with a user-supplied password.

Parameter values in CVL are lists of one or more strings. Each list element can be either a quoted string constant, a parameter, or a map. A map is a set of key-value pairs. The map notation eliminates the need to remember the ordering of values, and makes the meaning apparent. Map keys are string constants, and values can be string constants, other parameters, or even other maps. A map is just syntactic sugar for defining a string: for example, the map {a=>"b", c=>"d"} evaluates to "a=b&c=d". In the string representation of a map, non-alphanumeric characters in the keys and values are escaped, allowing for recursive map structures.

Let us now look at how the configuration parameters in the Groupware VAN are defined. The domain name, to be specified by the user for the entire network, is passed onto both the DNS and OpenLDAP appliances. The firewall appliance has a services parameter, which specifies the services that the firewall should allow and hosts to which those services should be forwarded. The Groupware VAN declaration puts the addresses and ports of the DNS and LDAP appliances into this parameter, via a map, in order to expose those services. This example illustrates how a VAN may have fewer parameters than the sum total provided by its components. The components may share common parameters, and values from one appliance can be used to configure another.

Disks in VM Appliances

Users are not allowed to make changes to the installed software in VM appliances, but of course must be able to modify their data. When updating a VAP, we must preserve the user's data. Our solution is to store user data either outside of the appliance by

using a network file system or on a separate disk dedicated to storing user data in the appliance. Appliances that need to access existing user files would likely want to use a network file system for user data. Service appliances, or appliances whose data is of no use outside of the appliance, would likely opt for the second option of a dedicated data disk in the appliance, as it introduces fewer dependencies.

Each virtual disk in a VM appliance is used either for storing appliance software or for storing user data. Disks storing appliance software are called *program disks*; they define the operation of the appliance. Disks storing user data are called *data disks*. All data disks, each described by a device name and an initial disk size, must be listed in the datadisks parameter inherited from VMAppliance. For example, Figure 2 illustrates a DNS appliance with a 100 megabyte data disk as device ide0:1; the data disk stores zone files for the appliance.

When updating an appliance, contents of the program disks are updated, but data disks are untouched. This allows the user's personal settings and data to persist across updates of the appliance.

Network Topology

A VANAppliance allows one or more appliances to be grouped into a single VAN appliance. This results in a strictly hierarchical network of virtual appliances. More general topologies are supported by CVL but, for simplicity, are omitted from this paper.

All components of a VAN are connected to the same virtual Ethernet network, which can be attached, via a gateway appliance, to another virtual network or the Internet. The gateway typically implements firewall, routing and NAT functionality.

The Interface ethernet0 declaration in VMAppliance guarantees that every VM appliance has a virtual interface. It is possible for a VM appliance to declare additional interfaces. The Collective Utility is responsible for assigning MAC and IP addresses to each of the interfaces in a VM appliance. Components in a VAN are connected to the same Ethernet segment via their VAN network interfaces, specified by the vanIF variable.

A VAN specification wishing to export a network interface must set the vanIF variable to an interface of one of its constituent appliances. For example, the Firewall appliance in Figure 2 has two interfaces, ethernet0 and ethernet1. Its ethernet1 interface, f.ethernet1, serves as Groupware's network interface. This allows the Groupware appliance to be connected as an appliance to another network (such as another VAN or the outside world). Groupware's defaultroute is set to f.ethernet0.ip so that all packets from Groupware's appliances are routed through the firewall.

Dependencies Between Appliances

Just as virtual machines can be started and stopped, so can VANs. Because services have dependencies, we have to start and stop them in a specific order. Appliance publishers list the services that their appliance provides and the services that their appliance needs for its operation. The Collective uses this information to construct a boot order and a shutdown order.

Every appliance inherits two variables from the Appliance object: provides and requires. These variables contain a list of strings representing the services provided or required by this appliance, respectively. For example, in Figure 2, the DNS appliance sets the provides variable to "DNS", and the OpenLDAP appliance sets the requires variable to "DNS". A VAN containing these two appliances would start the DNS appliance before the OpenLDAP appliance.

Repositories

A repository provides a location where a publisher can post successive versions of an appliance and users can find them. This section explains what repositories are and how they are used by publishers, users, and the Collective Utility.

Each Collective appliance repository holds the versions of a single appliance; the versions are numbered using integers starting from 1. Once a version has been written to the repository, that version becomes immutable. Each version of an appliance has a CVL file. For VM appliances, the VMware virtual machine files (.vmx, .vmdk, and .vmss) are also stored. To save time, disk space, and bandwidth, the virtual disks typically contain only the changes from the previous version of the appliance.

Publishers create and update repositories through the UNIX Collective User Interface command (cui for short). The publisher runs the command

```
cui create <repository>
```

to create an empty repository at the file path repository. The publish operation

```
cui publish <repository> <cvl>
```

stores the files representing a virtual appliance as the latest version of the appliance in the repository. For all appliances, this involves copying the CVL file into the repository. For a VM appliance, the VMware configuration file contains a list of all virtual disks comprising the VM appliance, and the CVL file designates some of the virtual disks as data disks. Virtual disks not designated as data disks are assumed to be program disks. The publish operation copies the contents of the program disks to the repository but does not copy the contents of data disks. This means that an appliance repository only contains appliance software and does not store any data disk content.

A repository can be hosted anywhere in the file system where a user can create a subdirectory. We access and store our repositories through SFS [9], which provides a secure access to a global namespace of files.

The Collective Utility

The Collective Utility manages both virtual appliances and hardware. The utility executes requests to start, stop, and update VAPs from users, and answers queries on the status of VAPs. It allocates hardware and network resources to VAPs and configures VAPs to use those resources.

The Collective Utility consists of a central cluster manager service and a host manager service for each computer in the cluster. The cluster manager accepts appliance management requests from users, decides on the allocation of resources across the entire cluster, and interfaces with the host managers, which are responsible for executing appliances on the respective hosts. The cluster manager also keeps track of the "truth" in the system, including the list of physical resources in the system, the VAPs that have been started, and the resources allocated to them. This information is stored on disk to survive cluster manager restarts.

The utility administrator is responsible for registering all the resources in the system with the cluster manager, so that it can keep track of the available resources and perform resource allocation to appliances. Using a command line tool, the administrator can register a host, specifying its resources – memory size and the maximum number of VMs hosted at any one time. In our prototype, we require each registered host have Red Hat Linux 9, VMware GSX Server 2.5.0, and the Collective software installed. The administrator also registers a set of VLAN numbers and public IP addresses with the cluster manager. These VLAN numbers and public IP addresses are assigned to virtual networks and to network interfaces connected to the public Internet.

The utility administrator can restrict the appliances the utility runs by providing it with a *blacklist* of repositories and versions that should not be run. For example, the administrator may wish to place all appliances with known security vulnerabilities on the list. The utility will not start new appliances that use versions on the blacklist. However, the utility will not stop already running appliances that violate the blacklist; instead, the administrator can query the utility for these appliances. The administrator can then either ask users to update or can forcibly stop the appliances.

Before using the utility, the user must first create a new appliance by creating a CVL file that inherits from the appliance to be run. As part of writing that CVL file, the user sets the values of the parameters of interest. The following is an example of a CVL file created by a user for the Groupware appliance:

```
import {
  url => "http://virtualappliance.org/Groupware",
  cvl => "Groupware.cvl",
} Groupware;

AcmeGroupware extends Groupware {
  domain = "acme.org";
}
```

The user can then use the utility to start, stop, and update the appliance. Below, we describe each of the available user commands in more detail and overview their implementation.

Starting a VAP

The command cui start <CVL> starts the appliance as specified in the <CVL> file. We first discuss how we handle virtual networks and then describe the implementation of the command.

Virtual Networks

Our design allows the component VM appliances in a VAN be run on one or more machines. Each running VAN has its own Ethernet segment, implemented as a VLAN (Virtual Local Area Network) on the physical Ethernet. All VM component appliances of a VAN on each host are connected to a dedicated VMware virtual switch on the host, which is bridged to the VAN's VLAN. Physical Ethernet switches that recognize and manage VLANs may need to be configured to pass traffic with certain VLAN tags to certain hosts. Since our experimental setup uses switches that ignore VLAN tags, no configuration is required.

The Collective also takes over the chore of assigning IP addresses to appliances. Each VAN is assigned a subnet in the 10.0.0.0/8 site-local IP address range, with the second and third octets of the address derived from the VLAN tag. So, each VAN has 256 IP addresses. Each virtual Ethernet adapter in each VM appliance is given a unique MAC address and an IP address from the pool of VAN's IP addresses.

In the case of sub-VANs, the internal interface of a gateway on a sub-VAN is assigned an IP address from the sub-VAN's IP address space. The external interface of the gateway is assigned an IP address from the address space of the parent VAN. Exported interfaces that do not connect the VAN to another VAN are given public IP addresses from a pool of IPs.

We use network address translation (NAT) to help route traffic between VANs and their parent networks. We must use NAT between the public Internet and our VANs since we assign site-local addresses to our VANs. Even though each VAN has a distinct site-local range, we still use NAT between VANs and sub-VANs to avoid setting up routing tables. For this reason, a VAN's chokepoint appliance, such as a firewall or router, should provide NAT functionality.

Implementation

To implement the start command, the cluster manager parses the CVL file. It imports CVL files from repositories where necessary, remembering the version number. It sets up the VAP's disks and then, if it finds that the VAP's requirements can all be satisfied, brings up the VAP. Note that a VAP may be a VAN whose components may themselves be VANs. From now on, we use the term *component VM appliances* of a VAP to refer to all the VM appliances defined by a VAP, its components, its components' components and so forth.

In the first step, the cluster manager sets up the program and data disks for all the component VM appliances in the directory containing the CVL file. Every component VM appliance is given its own subdirectory. The manager creates a new copy-on-write demand-paged version for each program disk, and if a specified data disk does not already exist, an empty data disk of the size specified in the CVL file is created. The appliance is responsible for detecting an all-zero data disk on boot and initializing it appropriately.

In the second step, the cluster manager ensures that all the required services are available, required parameters set, and required resources reserved. It generates a dependency graph from the provides and requires variables of all component VM appliances, and propagates parameter values to all the CVL files. For fault tolerance reasons, the cluster manager determines which resources are available by computing all the resources currently in use by all the running VAPs. It then decides where each VM appliance is to be hosted and reserves the memory requested in the appliance's .vmx files. Next, the cluster manager reserves a VLAN for each subnet and an IP address for each VM Appliance.

In the third and final step, the cluster manager brings up the VAN. It first sets up the hierarchy of networks by instructing all participating host managers to allocate VMware virtual switches and bridge them to the appropriate VLAN. It then starts up the component VM appliances, possibly in parallel, in an order satisfying the dependencies. The VMware Scripting API [24] is used to pass a VM appliance its parameters, including the assigned MAC and IP addresses, and to power it on. As soon as an appliance signals that it has successfully started, the cluster manager starts any appliances whose dependencies are now satisfied.

Stopping a VAP

The command cui stop <CVL> [<comp>] stops the entire appliance defined in the <CVL> file if no <comp> is given, otherwise it stops only the component appliance <comp>. As in CVL, components are specified using a dot-separated path, for example sub-component f of component g is written g.f.

Stopping a virtual appliance is more straightforward than starting one. Component VM appliances are stopped in the reverse order of startup. To stop a VM appliance, the cluster manager uses the VMware Scripting API to instruct the virtual machine to stop. VMware passes the stop request to a daemon running inside the appliance, which then initiates a clean shut down. If the appliance does not shut down within three minutes, VMware forcibly terminates it.

Updating a VAP

The command cui update <CVL> [<comp>] updates the entire appliance if no <comp> argument is given, otherwise it updates just the component <comp>. To minimize disruption, we do not require that all the VM appliances be shut down to update a VAN; only the affected VM appliances are. The cluster manager automatically derives the actions necessary to update an old version to the new by finding the differences between the two.

The cluster manager re-parses the CVL file, and simulates its execution to determine what the final state of the VAN should look like. It then finds the difference between that final state and the current VAN state, and determines the list of actions to transform the current state to the desired final state. The actions include:

- Starting an appliance that is present in the final state but not present in the current VAN state.
- Removing an appliance that is present in the current VAN state and not in the final state. First, it stops the appliance and the appliance data is moved to an attic directory. This prevents conflicts with any new appliance in future updates that might be given the same name as the removed appliance.
- Updating a VM appliance if its version has changed. This involves first stopping the VM, copying over the new program disks, and restarting the appliance. If an update requires data disks be modified, the new version of the appliance should include logic that, on boot, detects whether the user data is in the old format and, if so, migrates the data to the new format.
- Resending parameters to a VM appliance, if any have changed. This is done using the VMware Scripting API. An appliance wishing to respond to changes would run a daemon that checks for changes in the parameters and reconfigures the appliance appropriately. For example, when parameters are resent to a DHCP appliance, the daemon rewrites the DHCP configuration file and restart the DHCP server.

Costs of Virtual Appliances

The benefits of isolation and ease of management obtained from using virtual appliances are not without costs. First, starting up an appliance requires booting its operating system, which takes much longer than starting a service on an already booted operating system. Note, however, that this same procedure will bring up a pristine copy of the latest software on any machine. There is no extra cost associated with provisioning a machine to run a new service, re-installing the operating system to fix an error or updating a software to a new version.

Virtual appliances also have higher disk and memory storage requirements and slower execution due to virtualization overheads. Fortunately, virtual machine technology continues to improve. Recent work on VMware ESX Server improves the utilization of physical memory by adapting the allocation of physical pages dynamically and sharing identical memory pages across virtual machines [25]. Our previous work has shown

that demand paging and copy-on-write disks significantly decrease the cost of storing and sending updates using disk images [16]. Finally, it should be noted that our approach is not limited to virtual machines; with SAN adapters and appropriate management support from the computer's firmware, virtual appliances could be booted on raw hardware.

Experimental Results

We describe using the Collective system to build the following appliances: a Groupware virtual appliance network, a software development appliance, and Windows desktop appliances.

The Groupware Appliance

We created a groupware virtual appliance that provides a base Internet infrastructure for a small collaborating group. The plone web-based content management system was built with Debian GNU/Linux 3.0, and all the other appliances were built with Red Hat Linux 8.0. Open source software was used for the services. The software packages for the services and the versions used are shown in Figure 4.

Configuration and Deployment

Since the groupware bundles together mostly-configured appliances, this reduces the amount of configuration effort necessary for each instantiation. With the goal of building an infrastructure for a small group, we were able to set most software configuration options to reasonable defaults. For example, even though the OpenLDAP and Mail (Postfix and Courier) applications have a large number of configuration options, the corresponding appliances in groupware need only 12 and 9 parameters, respectively. In addition, some appliances share the same parameters. As a result, although a sum total of 30 parameters need to be set individually in the groupware appliances, deploying the whole suite of groupware services requires setting only 14 parameters. Figure 5 gives a complete list of all the parameters for the groupware appliances, along with a short description of each of them. The variables and their relationships are all specified in the CVL file, excerpts of which are shown in Figures 2 and 3.

The appliances use these parameters to configure themselves. Configuration works especially well for

Appliance	Software Package	Function	Size	Depends on		
				DHCP	DNS	LDAP
Firewall	iptables-1.2.6a	Forwards traffic only to group services	495 MB	---	---	---
DHCP	bind-9.2.1	Provides IP addresses to appliances	501 MB	---	---	---
DNS	dhcp-3.0pl1	Serves names of group services	501 MB	Yes	---	---
LDAP	openldap-2.0.25	Authentication and user database	578 MB	Yes	Yes	---
Mail	postfix-1.1.11 courier-0.42.2	SMTP, IMAP, and POP server	627 MB	Yes	Yes	Yes
Plone	plone-1.0.2	Web-based content management system	441 MB	Yes	Yes	---

Figure 4: Properties of appliances in the Groupware appliance.

Parameter Name	Appliances Using Parameter	Parameter Description
hostname	DHCP, DNS	Used to auto-populate the dhcpd.conf and DNS zone files.
domain	DHCP, DNS, LDAP	The DHCP server uses the domain parameter to set the proper domain name of the appliance. The LDAP server derives the root DN from the domain name.
rootdnpass	LDAP	The root DN password.
proxypw	LDAP, Mail	A proxy user is used for LDAP client authentication. This is the proxy user's password.
smtppw	LDAP, Mail	The smtppw is the password of the user who can read a user's Maildir parameter in the LDAP server.
country, state, city, org, ou, certemail	LDAP, Mail	These parameters are used to create X.509 certificates.
defmailhost	LDAP	The default mail host the SMTP server should deliver mail to.
forwarders	DNS	DNS caches to forward queries to.
rootpw	All appliances	The Unix root password.

Figure 5: Parameters of the Groupware appliance.

UNIX, where most configuration is done using text files and it is easy to interpose on the boot sequence. In our appliances, the boot sequence is modified to read in the appliance parameters and generate configuration files based on them, before appliance software is started. Typically, these configuration files are generated by using a template and filling in values based on appliance parameters.

For example, the firewall appliance has a parameter for specifying the ports and addresses of all the groupware services, and it configures itself to accept traffic only to those services. The DHCP appliance has parameters for specifying the MAC addresses, IP addresses, and hostnames of all the groupware appliances, and it transforms these parameters into a DHCP server configuration file. After rewriting the DHCP configuration file, the appliance restarts the DHCP server. Other appliances configure themselves in a similar fashion.

Characteristics of Groupware

As shown in Figure 4, the sizes of all the VM appliances are around 500 MB. This is large compared to the sizes of the services themselves. However, since we use SFS, during an appliance start-up only the required parts of the appliance are demand paged in. Furthermore, techniques exist (see the previous section) for decreasing appliance disk space requirements.

We timed the groupware start-up and shut-down on five 2.4 GHz Pentium 4 client machines with 1 GB of RAM each. The program and data disks were stored on a sixth 2.4 GHz Pentium 4 machine which acted as the SFS server. The clients and server were connected via 100 Mbps Ethernet.

The start-up time was 6.3 minutes with cold caches at the SFS server and clients and averaged 5.0 minutes over three runs with a warm cache. The shut-down time averaged 2.2 minutes over three runs. During start-up, the cluster manager took on average 1.1 minutes to set up appliance disks, reserve resources, and set up virtual networks. The remainder of the time is spent starting up the component VM appliances. The Red Hat 8.0-based component appliances took on average 1.8 minutes each to start before optimization. After removing unnecessary services from start, the appliances took about one minute to start. Currently, of that minute, the network start script takes 20 seconds; by taking advantage of the virtual nature of the network interface, we believe it can be optimized to under one second.

With further optimizations, appliances with smaller disk space and boot times could be built. For example, the floppyfw Linux distribution [22] combines both DHCP and firewall functionalities in a single 1.44 MB floppy disk that boots in less than 20 seconds in a virtual machine.

Software Development Environments

Software development today typically relies on a large number of tools and often specific versions of those tools. Acquiring all the tools required to develop and build a piece of software can take a lot of time. Appliances allow complex software to be used immediately and easily by the user by bundling all required tools and libraries into the appliance. We expect this will encourage more users to experiment with software and to participate in its development; for example, an appliance made up of free software could include the source and tool chain so that users could fix bugs and add features.

We have created an appliance for the SUIF research compiler infrastructure [1], a system that has been used by research teams both in and out of Stanford. Over the years, new researchers have spent hours setting up their environment to build the system before they could start experimenting with it. For example, SUIF is compiled and tested using the GCC 2.95 compiler and header files; users of Red Hat Linux 8 and 9 use GCC 3, and users of these systems would need to install the older compiler and headers. Even then, they would probably need to modify the build process to point SUIF to the alternate compiler and headers. In sum, we can avoid a lot of trivia by distributing the entire tool chain as a unit, which we did when we made SUIF into an appliance.

Users of the SUIF appliance and other desktop appliances may wish to mount their user files. We expect that users may want to access their existing files from the SUIF appliance, and therefore the SUIF appliance stores user files on a network file system rather than on a data disk. However, we feel it would be cumbersome if the user had to repeatedly enter their username, location of their files, and password into each appliance.

To mount user files into an appliance, we give the appliance a list of ways to mount the user files. Each way includes a URL and credentials; the URL tells the appliance how and what to mount and the credentials tell it who to mount as. The list is ordered. As the appliance traverses the list, it checks if it supports the protocol indicated in the URL. If it does support the protocol, it looks at the credentials. If it supports the authentication method indicated in the credentials, it tries to mount the file system. If it succeeds, it sets the home directory in the user record to the mounted file system. If it fails, it tries the next way of mounting the files.

Others have grappled with the problem of mounting user files across multiple operating systems. In Windows networks, *domains* authenticate users and provide roaming profiles; many other systems have similar notions. Recognizing the diversity in today's network sites, rather than forcing any specific system, we try to configure appliances with user files in as generic a fashion as possible.

Windows Appliances

To investigate the feasibility of Windows virtual appliances, we created virtual appliances reflecting common Windows desktop environments. The first

subsection describes the base of software we used in our experiments. While building these appliances, we had to deal with the peculiarities of Windows. In particular, we take a different approach (described in the second subsection) to configuring a Windows appliance since system properties, like the computer's *Security Identifier* (SID), do not reside in simple text configuration files as on Linux. Additionally, updating an appliance requires customizations be stored separately from programs, and this is difficult to enforce on Windows, but in the final subsection we observe that most applications partition themselves nicely.

The DesktopNet Appliance

We created two Windows virtual appliances; each virtual appliance runs a single application. We created one for Office 2000 and another for Internet Explorer 6, both running Windows 2000 Professional. To share program data and configuration settings between programs in different appliances, all appliances are configured to use roaming profiles and mount network shares from a central Samba server.

The Samba appliance runs Samba 2.2 on Redhat 8.0. It is configured to serve profile data and network shares off of its data disk, which starts off empty. Also, to add a user, the administrator must currently log into the Samba appliance and add the account manually. Alternatively, the Samba appliance could be configured to require an LDAP appliance for authenticating accounts.

Since Samba allows program data and configuration to live outside of the appliance, desktop applications generally require little direct configuration. The only required configuration variables for the desktop appliances are for making it talk to the Samba server. For example, Samba requires credentials for both the user (to access their files) and the machine (to join/authenticate to the domain). Appliances receive domain configuration (domain name and domain user/password) on-the-fly from the CVL specification, and users must log in to their desktop appliances before using them.

Configuring a Windows-Based Appliance

Under UNIX, appliance configuration involves taking in parameters supplied by the Collective and rewriting text configuration files residing in the appliance. Under Windows, however, changing such settings may require the modification of undocumented registry entries and a reboot of the appliance. Therefore we use published and tested tools where possible; our current approach for configuring appliances uses Microsoft's System Preparation Tool sysprep. Parameters configured through the appliance's CVL file are propagated to a script running in the appliance on boot, which prepares a sysprep unattended install file (sysprep.inf) with the passed-in parameters, and then initiates sysprep in the appliance. This requires a series of reboots to get the appliance into the appropriate state, but using sysprep guarantees that Windows-specific machine IDs are properly changed in each copy of the appliance.

Updating a Windows-Based Appliance

Normally, updating an appliance means replacing the appliance's program disks with updated ones. As long as data and user configuration reside on a separate data disk, no data is lost in this update process. Unfortunately, some Windows applications' use of the file system and registry can make separating program state onto a dedicated data disk difficult – any data stored under the application's Program Files directory or in system-wide registry entries could get blown away during an update.

Fortunately, many Windows applications store per-user configuration in the user's roaming profile directory (either through special per-user registry sections, which get backed by the roaming profile, or in the user's Documents and Settings folder, which also resides in the user's roaming profile). This behavior is actually mandated by Microsoft's Logo certification program so that roaming profiles work, and we have observed correct behavior in practice by doing updates on our desktop appliances.

For example, we made user customizations in Office 2000 (e.g., adding keyboard macros, dictionary entries, etc.) and updated from Office 2000 to Office 2000 Service Pack 3 by completely replacing the program disk; we observed that our user customizations remained intact. Likewise, we made similar customizations to Internet Explorer 5 (e.g., bookmarks, proxy settings, homepage, toolbar settings, etc.) and observed that these too were recognized after a complete overhaul of the publisher disk to Internet Explorer 6. Thus, although Windows applications can be more difficult to coerce into using particular parts of the file system, many turn out to be well-behaved on their own.

As illustrated by the experiences described in this section, we have encountered challenges in both configuring and updating Windows appliances. And although they present interesting problems, we have demonstrated techniques for turning Windows applications into parameterized virtual appliances.

Related Work

Package tools like Depot [5] and Debian's apt simplify the maintenance of individual software packages. Unlike most packages, appliances bundle groups of applications and provide a unified configuration interface. Also, virtual appliances running on the same computer are better isolated from each other than packages installed on the same computer. Tools like Sasify [13, 18], as well as the aforementioned package tools, simplify the task of keeping up with patches and software updates. Virtual appliances update the entire system, thus providing more assurance in the way of overall system testing at each version.

Like the Collective, Sun's N1 [21], HP's Utility Data Centre [8], and VMware's Control Center [23]

aim to automate managing hardware and software. The customer of the utility provides a higher-level description of the services they want to run and the performance they want from those services, and the utility decides how many servers to instantiate and what hardware to assign them to. The utility dynamically monitors load and can reassign computers from serving one service to another. The utility can also work around failures of hardware by booting a new instance of the service on good hardware.

Grid computing, of which Globus [7] is the leading example, automates the harnessing of thousands of computers across the world for running scientific code. The grid software for managing processes on thousands of machines may be useful for non-scientific code too. As a result, it seems that grid and utility computing are merging in some parts; Globus has recently expanded to web services.

Our research is focused on reducing software system administration cost by amortizing the configuration, installation, and update efforts over a large number of users. This led to the development of VANs, a way of distributing software that can be configured to suit individuals' needs and can be updated automatically.

Web application servers, like JBoss and Websphere, automate the deployment and management of services written in Java across a cluster of machines. They require software to be written to Java APIs. In contrast, our approach of using virtual x86 machines can be applied to most existing software.

Storage-area networks (SANs) and disk imaging [3, 14] have been used for years to reliably configure computers. Virtual appliances use disk imaging to predictably deliver updates to virtual disks. VMMs allow us to implement SAN-like capabilities in software without modifying hardware or the guest OS.

CFengine [4] is a tool for configuring operating system images from a central description. Like CFengine, CVL strives to describe what the state of a network of machines should be rather than describe the steps for configuring the network. Unlike CFengine, CVL passes information to appliances through a generic key-value pair interface and leaves the details of configuring individual nodes in the network to the appliance publishers.

Like the Desktop Management Task Force's (DMTF's) Common Information Model (CIM) [6] and SNMP MIBs [15], CVL describes objects with sets of properties. Unlike CIM and SNMP, the focus of CVL is not returning status and statistics but configuring objects, specifically networks of virtual appliances. To reduce duplication of values when configuring objects and to provide intelligent defaults, CVL has constructs for propagating one value to many parameters.

Conclusion

This paper develops the concept of virtual networks of virtual appliances as a means to reduce the cost of deploying and maintaining software. We have presented a language for specifying virtual appliances and algorithms for implementing them. The language is designed to allow user customization while supporting automatic updates. The Collective prototype we developed assigns virtual appliances to hardware in the system automatically and dynamically. The prototype uses repositories to find up-to-date versions of appliances.

We have shown how VAPs can be used to create a complete Groupware appliance that can be instantiated anywhere, a software development appliance that reduces the overhead associated with working on new software, and discussed how the approach can be used to create Windows-based appliances.

Our approach makes the management of software independent of the number of computers running the software. By placing the burden of maintenance on the appliance publisher, this approach makes it easier for users to run new computer software and to keep their systems up to date. The Collective Utility can even be used to disallow the execution of vulnerable software. This would eliminate incidents like the one where Microsoft itself was compromised by the Slammer worm when it failed to locate all vulnerable servers [2].

For information about releases of the Collective, please visit our web page at http://collective.stanford.edu/ .

Acknowledgements

This material is based upon work supported in part by the National Science Foundation under Grant No. 0121481 and Stanford Graduate Fellowships. We thank our shepherd Gerald Carter, Satoshi Uchino, and Will Robinson for their comments on the paper.

About the Authors

Constantine Sapuntzakis is currently pursuing a Ph.D. in Computer Science. To avoid the stresses of grad school, he likes to play system administrator for his research group and at home. He wants to make computer systems easier to use.

David Brumley is a Ph.D. student in Computer Science at Carnegie Mellon University. Previously, he was the computer security officer for Stanford University, where he responded to over 1000 incidents and authored such programs as the remote intrusion detector (RID) and SULinux (Stanford University Linux). David received his Bachelor's degree in Mathematics from the University of Northern Colorado and his Master's degree in Computer Science from Stanford.

Ramesh Chandra is a Ph.D. candidate in Computer Science at Stanford University. He received his B.Tech from the Indian Institute of Technology, Madras, India, and his M.S. from the University of Illinois at Urbana-Champaign, both in Computer Science. He is interested in systems research in general, and in particular operating systems, networking, and distributed systems.

Nickolai Zeldovich is a Ph.D. student in Computer Science at Stanford. He received his Bachelor's and Master's in Computer Science from MIT in 2002. In his spare time, he likes to windsurf, hack on OpenAFS, and collect old computer equipment.

Jim Chow is currently a Ph.D. student in Computer Science at Stanford University. His interests include operating systems and systems security. He graduated in 2000 with a BS in Electrical Engineering and Computer Science from UC Berkeley.

Monica Lam is a Professor of Computer Science at Stanford. She received a Ph.D from Carnegie Mellon University in 1987 and a B.S. from University of British Columbia in 1980. Her current research interests are in improving software productivity and usability of computers. Honors for her research include an NSF Young Investigator Award and an ACM Most Influential Programming Language Design and Implementation Paper Award.

Mendel Rosenblum is an Associate Professor of Computer Science at Stanford. His contributions to the field of systems, including the Log-structured File System, SimOS, Hive, Disco, and VMware, earned him the ACM SIGOPS Mark Weiser award in 2002.

References

[1] Aigner, G., A. Diwan, D. Heine, M. S. Lam, D. Moore, B. Murphy, and C. Sapuntzakis, *An Overview of the SUIF2 compiler Infrastructure*, http://suif.stanford.edu/suif/suif2/doc-2.2.0-4 .

[2] Associated Press, *Microsoft also gets slammed by worm*, http://www.cnn.com/2003/TECH/biztech/01/28/microsoft.worm.ap/ , January, 2003.

[3] Barnett, T., K. McPeek, L. S. Lile, and J. Ray Hyatt, "A web-based backup/restore method for Intel-based PC's," *Proceedings of the 11th LISA Conference*, pp. 71-78, October, 1997.

[4] Burgess, M., "A Site Configuration Engine," *USENIX Computing Systems*, Vol. 8, Num. 2, pp. 309-337, 1995.

[5] Colyer, W. and W. Wong, "Depot: A tool for managing software environments," *Proceedings of the 6th LISA Conference*, pp. 153-162, October, 1992.

[6] *Desktop Management Task Force – Common Information Model*, http://www.dmtf.org/standards/standard_cim.php .

[7] Foster, I., C. Kesselman, J. M. Nick, and S. Tuecke, "Grid Services for Distributed System Integration," *IEEE Computer*, Vol. 35, Num. 6, pp. 37-46, 2002.

[8] *HP Utility Data Center*, http://www.hp.com/solutions1/infrastructure/solutions/utilitydata/ .

[9] Mazières, D., M. Kaminsky, M. F. Kaashoek, and E. Witchel, "Separating key management from file system security," *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP '99)*, Kiawah Island, South Carolina, December, 1999.

[10] *Microsoft Security Bulletin MS01-020: Incorrect MIME header can cause IE to execute attachment*, March, 2001.

[11] *Microsoft Security Bulletin MS02-039: Buffer overruns in SQL Server 2000 resolution service could enable code execution*, July, 2002.

[12] Red Hat, *Red Hat Linux 8.0: The official Red Hat Linux security guide*, http://www.redhat.com/docs/manuals/linux/RHL-8.0-Manual/security-guide/ .

[13] Ressman, D. and J. Valdes, "Use of CFengine for automated, multi-platform software and patch distribution," *Proceedings of the 14th LISA Conference*, pp. 207-218, December, 2000.

[14] P. Riddle. "Automated Upgrades in a Lab Environment," *Proceedings of the 8th LISA Conference*, pp. 33-36, September, 1994.

[15] Rose, M. and K. McCloghrie, *RFC 1212: Concise MIB definitions*, March, 1991.

[16] Sapuntzakis, C., R. Chandra, B. Pfaff, J. Chow, M. S. Lam, and M. Rosenblum, "Optimizing the Migration of Virtual Computers," *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation*, pp. 377-390, December, 2002.

[17] Sapuntzakis C. and M. S. Lam. "Virtual appliances in the Collective: A Road to Hassle-free Computing," *Proceedings of the 9th Hot Topics in Operating System*, May, 2003.

[18] Shaddock, M., M. Mitchell, and H. Harrison, "How to Upgrade 1500 Workstations on Saturday, and Still Have Time to Mow the Yard on Sunday," *Proceedings of the 9th LISA Conference*, pp. 59-66, September, 1995.

[19] *Slammer worm hits the Net*, http://news.com.com/1200-1001-982780.html , January, 2003.

[20] *Stanford's e-mail service restored following shutdown*, http://www.stanford.edu/dept/news/pr/03/virus611.html ,June, 2003.

[21] *Sun N1*, http://www.sun.com/n1 .

[22] *floppyfw*, http://www.zelow.no/floppyfw/ .

[23] *VMware Control Center*, http://www.vmware.com/products/cc_features.html .

[24] *VMware Scripting API*, http://www.vmware.com/support/developer/scripting-API/doc/ .

[25] Waldspurger, C. A., "Memory resource management in VMware ESX server," *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation*, December, 2002.

Appendix A: BNF Grammar of CVL

```
Program      ::= ProgramStmt*
ProgramStmt  ::= ImportStmt | AsgnStmt |
                 ObjDecl

ImportStmt   ::= import Map Item ;
ObjDecl      ::= Item extends Item { Stmt* }

Stmt         ::= VarDecl | CompDecl |
                 AsgnStmt
VarDecl      ::= var AttrList ItemList ;
AttrList     ::= QuotedStr*
CompDecl     ::= Item ItemList ;

AsgnStmt     ::= Ident = RhsList ;
RhsList      ::= Rhs | Rhs , RhsList
Rhs          ::= QuotedStr | Ident | Map
Map          ::= { PairList }

PairList     ::= Pair | Pair , PairList
Pair         ::= Item => Rhs

URL          ::= [file | http]://non-whitespace
QuotedStr    ::= "any characters (quotes escaped)"
Ident        ::= Item | Item . Ident
ItemList     ::= Item | Item , ItemList
Item         ::= alpha alphanumsym*
alphanumsym  ::= alpha | 0-9 | _
alpha        ::= a-z | A-Z
```

Generating Configuration Files: The Directors Cut

Jon Finke – Rensselaer Polytechnic Institute

ABSTRACT

The generation of system configuration files and other documents directly from a database has proven to be a very powerful technique. However, there were some limitations to this approach. With the introduction of Oracle 8i, and more specifically, the addition of support for XML, we have been able to eliminate many of these limitations and take the file generation and maintenance to a new level.

Introduction

At LISA 2000, I presented a paper [6] on generating system configuration files and other documents directly from data stored in a relational database. This approach had many strengths, such as allowing version control of documents and files, selective regeneration based on need (only regenerate files that have changed), centralizing the file generation logic. The mechanics of writing actual files was handled by a simple, generic file generation program and all of the hard work was done in the database. This model has worked well for us, and has proven to be quite flexible and convenient for the deployment of new files.

With all the strengths of this approach, we also hit some limitations. We found that the extraction of the data from the database required one particular skill set, while formatting that data for presentation or eventual end use required a different, often unrelated, skill set. Simply put, the extraction of the data required database manipulation skills, i.e., the ability to program in PL/SQL.[1] Formatting that extracted data required different skills and knowledge. This latter set of skills and knowledge could be, e.g.,

- The format and structure of OS configuration (e.g., /etc/passwd and /etc/group);
- typesetting skills (e.g., LaTeX, HTML); or
- how "corporate branding" (e.g., logos, templates) might affect web pages.

A second issue came up with the generation of web pages. As the visibility of web documents was growing, so was the interest by the "Marketing and Media Relations" folks in the actual appearance of the web pages, down to the use of graphics and buttons and the look and feel of the web pages. To make matters worse, this look and feel was constantly changing and evolving, and it was a chore to keep the generated web pages looking like the rest of the site's pages. Some of this could be handled via cascading style sheets, and some other tools to allow the graphics designers to specify sections of "boiler plate" within documents, but the basic structure and layout was still dictated by the PL/SQL code. Any change in what data elements were displayed, or even the order of columns in a table still required intervention by the programmers and that pleased neither the programmers nor the web folks.

Clearly, we needed a way to separate the extraction of the data and it's reformatting. This would let the database programmers concentrate on extracting the data without getting bogged down with the final format issues. Folks who were concerned with format, such as web designers, marketing managers, etc., could concentrate on the formatting process. With the release of Oracle 8i, support was added for the processing and transformation of *eXtensible Markup Language* (XML) [2] documents. This provided a method of extracting the data from the database in a display neutral format, and then allowing the web designers to apply whatever transforms and formatting that was required. This meant that once the file extraction routine was written, the PL/SQL programmers were done and the web designers could take over and make formatting changes as their needs and schedules dictated. In addition, since the database could invoke the translations internally, we were able to keep all of the version control and process monitoring [8] features that we found useful in the first version, and use the same generic program to actually write the files.

XML – A Brief Primer

The "parent" of XML is Standard Generalized Markup Language (SGML). Another offspring of SGML is HTML, which many of us are already familiar with. One of the problems with HTML, is that it is not extensible, or perhaps worse, it does get extended by some browsers and servers and not by others. I am certain that many of us have visited web pages that don't work quite right on our browser of choice. XML was designed to be extensible, yet contained so that applications developed to process an XML document, can deal with any XML document. A good discussion of this is available in Chapter 6 of *The LaTeX Web*

[1] PL/SQL is a procedural extension to Oracle that allows you to write programs that are stored and executed in the Oracle database itself [11].

Companion [9], including why XML will not run into this problem, unlike HTML.

But XML is not limited to document preparation. Another area where XML is used is in business to business applications to allow businesses to exchange information in a consistent manner. Inherent in an XML document, is the Document Type Definition (DTD). The DTD (or schema) provides a set of rules that not only defines the element names and types, but also the relationship between data elements in the XML document. This provides a way to validate an XML document to ensure that it is correct. This schema is in fact another XML document. By sharing these XML schemas, different parties can independently develop XML documents and interfaces that will inter-operate.

Using XML to Encode OS Configuration Files

But XML has more uses than just document preparation or exchanging information between business. We have the need to generate /etc/passwd files or the equivalent from our database and make it available to different kinds of systems. Although we could generate the traditional ":" delimited form of the file and then use tools and scripts to reformat that if needed (like for our LDAP servers), we face a problem if we need to include more information about an entry. We can add more fields, but we then risk breaking those tools and scripts. We can instead encode our password entry information in XML.

The XML version of our password file consists of two parts. The first, seen in Figure 1 is the DTD which defines how the rest of the file will be formatted. This can be very useful to people who want to process this data and allows us to mechanically verify the format of the data. The second part, seen in Figure 2 contains the actual password data enclosed by <etc_passwd> tags. Each record (or line in the /etc/passwd file), is in turn enclosed with the <pw_entry> tags. Within each record are a number of elements. The first six should look pretty familiar, but there are some additional fields that may be useful to other applications. We will see some examples of how these are used later on.

XML and Translation

One of the facilities provided with many implementations of XML, is the ability to use an *eXtensible Stylesheet Language* (XSL) [1] template (or style sheet) to transform an XML document into some other format such as HTML, LaTeX or whatever is needed. The important point being that this enable us to separate the extraction of the data from the database, from the details of the final output format, and in fact, the same XML data can be translated by several different templates to generate different types of documents.

```
<?xml version = '1.0'?>
<!DOCTYPE etc_passwd [
<!ELEMENT etc_passwd (pw_entry)*>
<!ELEMENT pw_entry (username?, unixuid?,
    gid?, gecos?, homedir?, shell?,
    person_id?, user_type?, platform?)>
<!ATTLIST pw_entry platform
                    CDATA #REQUIRED>
<!ELEMENT username (#PCDATA)>
<!ELEMENT unixuid (#PCDATA)>
<!ELEMENT gid (#PCDATA)>
<!ELEMENT gecos (#PCDATA)>
<!ELEMENT homedir (#PCDATA)>
<!ELEMENT shell (#PCDATA)>
<!ELEMENT person_id (#PCDATA)>
<!ELEMENT user_type (#PCDATA)>
<!ELEMENT platform (#PCDATA)>
]>
```

Figure 1: XML DTD for /etc/passwd.

In Figure 3, we have an XSL transform that will take an XML version of our password database, and create a conventional /etc/passwd file. Along with simply reformatting the data into the desired format, it also does a selection to obtain all of our base entries (our normal users), along with the special entries we want on our AIX hosts. It does this by comparing the platform attribute for the desired targets. A trivial modification to this selection, and we can generate the passwd file for other platforms as needed.

```
<etc_passwd>
   <pw_entry platform="BASE">
      <username>finkej</username><unixuid>58220</unixuid><gid>4000</gid>
      <gecos>Jon Finke</gecos><homedir>/home/20/finkej</homedir>
      <shell>usr/bin/session</shell>
      <person_id>91325789</person_id><user_type>PRIMARY-EMP</user_type>
      <platform>BASE</platform>
   </pw_entry>
   <pw_entry platform="BASE">
      <username>hudsod</username><unixuid>58221</unixuid><gid>4000</gid>
      <gecos>Dave Hudson</gecos><homedir>/home/21/hudsod</homedir>
      <shell>usr/bin/session</shell>
      <person_id>91325792</person_id><user_type>PRIMARY-EMP</user_type>
      <platform>BASE</platform>
   </pw_entry>
   . . .
</etc_passwd>
```

Figure 2: XML version of /etc/passwd.

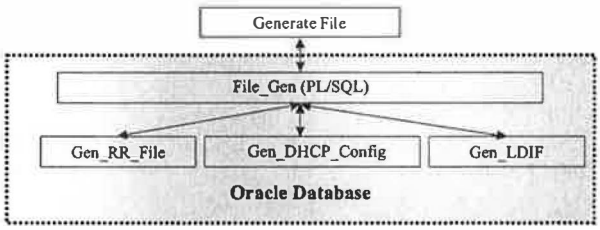


Figure 4: Original file model.

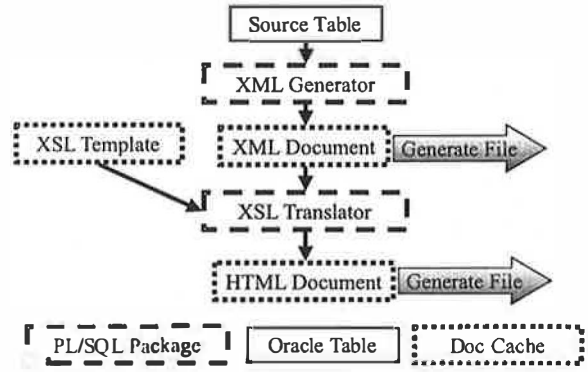


Figure 5: New file model.

Original Model

In the original model, seen in Figure 4, we would write a file specific PL/SQL package that "knew how" to generate the final file format. This package would be called by the File_Gen PL/SQL package that worked closely with the Generate_File program to extract the formatted information from the file specific package, and write it to the hosts file system (The Generate_File program can also read in files, read and write pipes, stdin, stdout and stderr). Together, these would handle most of the details of access control, error checking, version control and other details, allowing the file specific packages to concentrate on extracting the appropriate data and formatting it.

New File Model

As with the original model, we still write the appropriate PL/SQL code (the XML Generator in Figure 5) to generate a file from the desired data in the source table(s), only this time, instead of HTML or LaTeX, we generate it as an XML format document. We can, if desired, write this XML file out to an external file system, using the same Generate_File program as we used for the original model. All of the version control and selective generation features of the original approach are still available for use. The resulting XML files can then be used by the web developers to produce the desired outputs files, and made available for other services that need the data.

We don't need to stop with just the XML file. We can transform that XML document, inside of Oracle, with the appropriate XSL template into our desired HTML or LaTeX (or whatever) document and write out that file using the Generate_File program, just as we did before. This allows us to take full advantage of the version numbering of the original data, as well as the version number of the XSL template. One nice thing about this, is if the web developers provide a new style sheet via the XSL template, that forces the regeneration of all of the documents that used that style sheet. Note – this only requires that the original XML data documents be re-transformed, we do not need to repeat the original database queries used to generate them.

Version Control

In the original implementation, file version control was a one step process. We simply recalculated the version number based on the data in the database, and compared that to the version number in the file. If there was a difference, we regenerated the file with the new version number. With the addition of the XML support, we have at least two, and possibly three steps. As with the old model, we recalculate the version number based on the data, and then compare it with version number of the XML data stored in the document cache. At this point we now have an XML document, still stored in the database with a new version number. We can, if desired, write out this XML document to the external file system using the Generate_File program.

But although the XML file is nice, that is often not our desired objective. Instead of writing the XML document to a file, we can transform the XML document, via an XSL transform into a new document,

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="text" indent="no" media-type="text/plain" standalone="yes" />
  <xsl:template match="/">
    <xsl:for-each select="/etc_passwd/pw_entry[platform='BASE' or platform='AIX']">
      <xsl:sort select="unixuid" order="ascending" case-order="lower-first"/>
      <xsl:value-of select="username"/>:*:<xsl:value-of select="unixuid"/>:
          <xsl:value-of select="gid"/>:<xsl:value-of select="gecos"/>:
          <xsl:value-of select="homedir"/>:
          <xsl:value-of select="shell"/><xsl:text>&#xA;</xsl:text>
    </xsl:for-each>
  </xsl:template>
</xsl:stylesheet>
```

Figure 3: XSL transform for /etc/passwd.

possibly HTML or other format. This document is also still stored only in the database. It is however, a trivial matter to write this final form of the document to an external file. As an added twist, the version number of this transformed document is simply the greater of the version number of the base XML document and the version number of the XSL transform, which is yet another XML document stored in the database. In this way, when the XSL transform document is updated, all of the final documents that depend on that transform are now "out of date" and will be regenerated at the next available opportunity.

XML Document Cache

It quickly became obvious that we needed a way to store and reference XML documents easily within the database. Even if we were not concerned with the cost of regenerating the base XML documents every time we wanted to transform it, it is often desirable to be able to maintain a consistent data image. By generating and storing an XML document in the database, we are able to apply different transforms and know we are working on the same dataset.[2]

We had some specific applications in mind, and these helped drive our implementation. One of them was the our web directory of registered home pages. We actually have two different ones, the student home page directory and the Faculty/Staff home page directory. Aside from the title and the actual entries, these both use the same format. This was a case where we had the same schema but two different instances of that document. A second case was our departmental staff listing. For each department, we have a web page with all the people in that department. All use the same schema, but we have many different versions (one per department). In this case, we are looking at one instance (department staff list) of one schema (directory entry), but with many sub-documents, one for each department.

Another concern was maintaining access control on schemas, instances and documents. It is important to us to allow the web developers to update the templates used to generate the departmental directory web pages, yet not be able to change the templates used to generate the LaTeX used to produce the printed departmental directory. That access has to be granted to the text processing folks who manage the LaTeX world. Yet, both groups of people need to be able to read the same base documents. It is likely that our access control systems will become more fine grained over time.

Organizing Documents

The key element of any XML document is the schema. This defines what can be included in the document and how it is represented. Even if we don't actually validate the document, or even have the schema formally documented, it exists as a concept, and gives us a good starting point for organizing documents. In Figure 6 we have a schema, which would have basic information such as the name, a description, who created it and when, and so on. There still isn't any data at this level. In most cases, we will have a DTD document associated with this schema. This

[2]Storing an XML document in Oracle is trivial. Oracle has a data type CLOB that can handle a document up to 2 GB in size (assuming you have the disk space available to handle it in the database). The XML stored packages provided by Oracle [10] allow manipulation of XML documents as CLOBs.

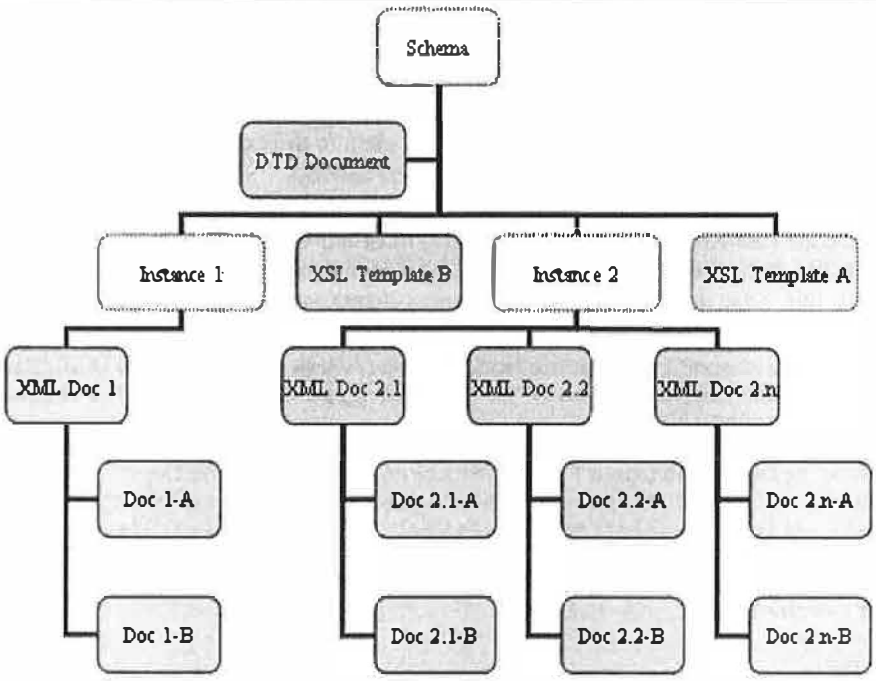


Figure 6: Schema model.

provides a guideline for both developers who want to generate XML documents as well as people who want to write XSL or other types of transforms.

Now that we have a schema, the next step is to define an instance of this schema. This will be an actual set of data, such as the password file or the departmental directory. This data will presumably need to be regenerated on a periodic basis. Since this is production data, we need to impose access controls on this instance of the schema; who can update it, and who can reference it. This instance may actually result in multiple sets of data – for instance, our employee directory is a set of documents, one per department. In this diagram, we have two instances, the first with just a single document and the second with multiple documents.

Since most people don't actually want to read XML files, the next step is to define an XSL template that will transform each of the XML data documents into the desired format, resulting in a second set of documents that can be written out as files or displayed via the web. We can provide additional translations to produce other formats. In the example of our staff directory, some departments want to have their own directory page on their web site. This allows them to use the official, most accurate, directory information and still maintain their desired look and feel. In Figure 6 we have two XSL templates (A and B), and we can apply those to each of the XML documents resulting in two output documents per input document.

These multiple transforms also allow us to include more data elements in the base XML documents that we might not want to include in the most broadly published documents. In the case of our employee directory, we include home address and telephone numbers, as well as cell phone and pager numbers. We do not want to put this into our regular online directory available to anyone via the Web, but we do want to include this in internal use department directories. By being able to control what elements are included as part of the XSL transform, all of our directory listings are able to have the most accurate and timely information.

Accessing Documents

When a document is stored in the document cache,[3] (see Table 1), we store a bunch of additional information about the document. This allows us to determine where a document originated, who put it there, and when. We also assign a unique document_id value that can be used to reference this document elsewhere in the system.

Extracting Documents

All documents stored in the document cache can be accessed via their Document_Id value. This allows us to have a couple of common routines used to extract

[3]My use of the word "cache" may be confusing to some folks. Caches are often short lived, whereas in this project, may last longer. An alternate description would be "Document Store."

Document_Id	Number	A unique identifier for this document
Schema_Id	Number	The identifier (primary key) of the schema that defines this document.
Instance_Id	Number	The identifier (primary key) of the schema instance that this document is part of. For a DTD document, this is the same as the Schema_Id.
Base_Doc_Id	Number	For a generated (XSL Transformed) document, this is the Document_Id of the base XML data.
Template_Id	Number	For a generated document, this is the document_Id of the XSL transform.
Doc_Code	varchar2(16)	For schema instances that have sub documents, this is the key that distinguishes the different documents from each other.
Doc_Type	varchar2(16)	This identifies the format of the document, such as "XML", "XSL", "DTD" and so on. This can be useful when displaying a document.
Doc_Title	varchar2(64)	This is a simple identifier for the document – useful for listing available documents without having to open each document up to determine what it is.
Description	varchar2(2000)	A place for a more in-depth description of a document than is allowed by the title field.
Entry_Method	varchar2(8)	A flag that indicates how this document was obtained, such as from an external URL, generated via direct query or other methods.
Ref_URL	varchar2(255)	The URL used to obtain this document if it was loaded via URL. Other entry_Methods may store source information such as the procedure or package name that generated the document.
Doc_Size	Number	The size of the document, in bytes.
When_Inserted	Number	The version number of this document, obtained for a system wide sequence number space.
Content	CLOB	The actual document, up to 2 GB in length.

Table 1: Document cache table.

a document, given the Document_Id. For both Generate_File and web display, we want to get the document one line at a time, until the end of the document. We did this with a PL/SQL function Get_Line as seen in Figure 7. This gets the document as a CLOB from the database, and breaks it out one line at a time.

```
Function Get_Line(doc_id in number)
return varchar2
is
    Next_Eol        Integer;
    Saved_Offset    Integer;
    Len             Integer;
begin
    if not Document_Open
    then
            Document_Open := True;
            Document := Get_Doc(Doc_Id);
            Document_Offset := 1;
    end if;
    next_eol := Dbms_Lob.Instr(Document,
             chr(10), Document_Offset);
    if next_eol = 0
    then
        Document_Open := False;
        return null;
    end if;
    Len := Next_Eol - Document_Offset ;
    Saved_Offset := Document_Offset;
    Document_Offset := Next_Eol + 1;
    Return Dbms_Lob.Substr(Document,
                    Len, Saved_Offset);
end Get_Line;
```

Figure 7: Get_Line function.

Generating XML from a Simple SQL Query

One of the handy PL/SQL packages that comes with Oracle 8i, allows you to define a SQL query, and pass it to a routine, and it will execute the query and return an XML document. Now it is possible that the existing Oracle table layout does not directly provide you with the exact thing needed for your XML, so one approach is to create an view.[4] To create the XML version of our password file, we created the view shown in Figure 8. This view extracts all of our normal userids (based on the SOURCE attribute) and then combines that list with the "SPECIAL" userids. For normal users, we derive the home directory from the UID and username, and add a default shell and a default group id if none is supplied. For the special ids, we generate those same fields if none are provided in the Special_Ids table.

The next step is to turn that into XML. To do this, we packaged up the Oracle routines, with some of our own to give us the code seen in Figure 9. We simply set up the query of the view, and then pass it to the XML_Cache_Maint package which will execute the query, turn it into XML, store it and return the Document_Id to us. This gives us the XML seen in Figure 2. The first bunch of parameters are used to tie into the document cache, defining the schema name and instance, adding a title and a description. Since we are providing this as a single document (no sub documents), there is no Doc_Code value. The next set of parameters are passed to Oracle and tell it how to label the XML. The gen_dtd tells it to include the DTD at the start of the document, and the last parameter tells the document cache to create a new schema if one does not already exist. This last argument is to help bootstrap the system into place and should not be needed for production file generations.

Generating XML from a Not So Simple SQL Query

The method of generating an XML document from a SQL query can handle a lot of basic XML generation needs, but at times, we need more complex documents. Consider the XML required to generate the /etc/group file (see Figure 11). The DTD for this (see Figure 10) is similar to that for /etc/passwd, but with the change that one of the elements (userlist) is in fact a subtype with multiple elements allowed.

As with the /etc/passwd example, the entire document is wrapped in a <etc_group> tag, and has a number of <group_entry> entries. The expected group fields (group name, group password, group id) as well as a <platform> attribute that we use to define the type of system for which this entry is intended. A group can have zero, one or more than one members in it. To handle these, we have the field <userlist> which will hold as many usernames (all tagged with <userlist_item> tags). You will note in the second entry for the group user, the user list entry is written as <userlist/>. This indicates an empty list.

[4] A view is a predefined query that can operate on one or more base table and be referenced much like a table.

```
Select Username, unixuid "UNIXUID", nvl(unixgid,4000) GID,
       Public_Personal_Info GECOS, 'usr/bin/session' SHELL,
       '/home/' || ltrim(to_char(mod(unixuid,100),'00')) || '/' ||
       username HOMEDIR, owner PERSON_ID, source USER_TYPE, 'BASE' PLATFORM
  from logins
 where (    substr(source,1,7) = 'PRIMARY' or source = 'SECONDARY')
Union
Select L.username, l.unixuid, nvl(l.unixgid,4000),
       L.public_personal_info,nvl(SI.Shell,'usr/bin/session'),
       nvl(SI.Home_Dir,'/home/' || ltrim(to_char(mod(L.unixuid,100),'00')) ||
                '/' || l.username), owner, source, nvl(SI.Platform,'BASE')
  from Logins L, special_Ids si
 where l.source = 'SPECIAL'
   and l.username = si.username
```

Figure 8: Oracle view: etc_passwd.

```
<!DOCTYPE etc_group [
<!ELEMENT etc_group (group_entry)*>
<!ELEMENT group_entry (gname, gpasswd?,
            gid, platform?, userlist?)>
<!ATTLIST group_entry platform CDATA
                        #REQUIRED>
<!ELEMENT gname (#PCDATA)>
<!ELEMENT gpasswd (#PCDATA)>
<!ELEMENT gid (#PCDATA)>
<!ELEMENT platform (#PCDATA)>
<!ELEMENT userlist (userlist_item)*>
<!ELEMENT userlist_item (#PCDATA)>
]>
```

Figure 10: DTD for /etc/group.

To generate this data, we need to access data from three tables. The primary table is the groups table that holds the basic information about the group (aside from the membership). We then need to reference the group_members table to get the list of members for each group and finally the logins table to get the actual usernames. Fortunately, Oracle provides us with a way of doing that.

To do this, we first define a type XML_Etc_Group_Userlist as a TABLE OF VARCHAR2(8). We then create an object view of the groups as shown in Figure 12. This creates a view of a complex object that includes a sub-query for each row. When the view is referenced, for each row (group), it obtains all of the members of that list, and converts those usernames. That entire list is returned as the single element userlist. When we pass a "SELECT * FROM XML_ETC_GROUP" query to Cache_Query routine, the XML in Figure 11 is generated.

Complex Generation

When the generation of the XML becomes too complex for the query method, even with the complex

```
Query := 'Select username, unixuid, gid, gecos, homedir,'
      || '       shell, person_id, user_type, platform'
      || '  from XML_Etc_Passwd order by uid';

did := XML_Cache_Maint.Cache_Query(
        query => Query,
        Schema_name => 'etc_passwd',
        Schema_instance => 'General RCS',
        d_title => 'ETC Passwd',
        d_code => Null,
        d_desc => 'An XML version of the user base',
        row_tag => 'PW_Entry',
        row_set_tag => 'etc_passwd',
        row_id_attr_name => 'Platform',
        row_id_attr_value => 'PLATFORM',
        gen_dtd => True,
        create_schema => True);
```

Figure 9: Create and cache XML from a query.

```
<etc_group>
   <group_entry platform="rs_aix">
      <gname>adm</gname><gid>4</gid><platform>rs_aix</platform>
      <userlist>
         <userlist_item>bin</userlist_item>
         <userlist_item>adm</userlist_item>
      </userlist>
   </group_entry>
   <group_entry platform="BASE">
      <gname>user</gname><gpasswd>*</gpasswd><gid>4000</gid><platform>BASE</platform>
      <userlist/>
   </group_entry>
  ...
</etc_group>
```

Figure 11: XML Data for /etc/group.

```
Select Group_Name, Group_Passwd, Group_Id, group_index, nvl(platform,'BASE'),
       cast ( multiset ( select username
                           from group_members gm, logins l
                          where gm.group_index = g.group_index
                            and gm.login_id = l.login_id
                  ) as XML_Etc_Group_Userlist
         ) as userlist
  from groups g
```

Figure 12: XML_Etc_Group view definition.

views, we can fall back to building the XML element by element. Although we could simply build it up as a string using character manipulation, Oracle provides some additional packages that can assist.

There are several approaches for manipulating XML documents, and one of those is the Document Object Model (DOM). With the DOM model, the XML document is represented by a tree structure built in memory. This can be manipulated in a number of ways, and then eventually exported as an XML document. Oracle has a Java implementation of the DOM routines, and they have provided a PL/SQL interface to them as well. In Figure 13, we have the main procedure that we use to generate our departmental directory.

This creates a new document using the DOM routines, assigns a root node to it, and then calling some existing directory code (Get_Univ_Web_List) we get a list of all departments in the directory. We pass each entry to a formatting routine (Gen_Entry – Figure 14) which formats each entry into a node on the XML tree. Since a given entry may have sub entries, we check each node for children, and append appropriate directory entries where needed. When we reach the end of the list of departments, we take the now fully populated XML tree and save it in the document cache. This XML document is now available for transformation into other formats or to be written as XML for external processes to manipulate.

Although this approach of building up an XML document one element at a time may appear to be tedious, it is actually less tedious than the previous approach of generating HTML directly. In addition, by moving the details of the presentation into the XSL transformation, the resulting XML code is simpler than the HTML it replaces. Another nice win with the generation of XML is that since the transformation can ignore extra data elements, it makes it much easier to write generic XML generation routines that include everything and let the XSL transforms sort it out. This eliminates a lot of duplicate or very similar code and ensures that logic changes apply everywhere.

```
doc := XMLDOM.newDOMDocument;   -- Get a fresh, empty document
root := XMLDOM.createElement(doc,'Institute_Directory'); -- Create a root node
dmy := XMLDOM.appendChild(XMLDOM.makeNode(doc),root);   -- And link it in
loop
        R := Generate_FacStaff_Dir.Get_Univ_Web_List;
        exit when r.name is null;
        Gen_Entry(root,R);
end loop;
did := XML_Cache_Maint.Cache_Dom_Doc(
        Document => doc,
        Schema_name => 'Department_Phone_List',
        Schema_instance => 'Institute Telephones',
        d_title => 'Institute Office Directory',
        d_code => Null,
        d_desc => 'Directory of phone nums/web pages for Inst. Offices.',
        d_type => 'XML');
Commit;
XMLDOM.freeDocument(doc);
```

Figure 13: Directory generation main procedure.

```
procedure Gen_Entry(node in XMLDOM.DOMNode, R in Generate_FacStaff_Dir.Univ_Tel_Rec) is
        This_Node       XMLDOM.DOMNode;
        dmy     XMLDOM.DOMNode;
        Child   Generate_FacStaff_Dir.Univ_Tel_Rec;
begin
       -- Create the entry node and add it to the list
        This_Node := XMLDOM.makeNode(XMLDOM.CreateElement(Doc,'Dir_Entry'));
        dmy := XMLDOM.appendChild(node,This_Node);
        -- Populate the top level records in the node
        XML_DOM_Utils.Make_And_Append(This_Node,'Name',R.Name);
        XML_DOM_Utils.Make_And_Append(This_Node,'TELEPHONE',R.Phone);
        XML_DOM_Utils.Make_And_Append(This_Node,'URL',R.Url);
        XML_DOM_Utils.Make_And_Append(This_Node,'EMAIL',R.Email);
        XML_DOM_Utils.Make_And_Append(This_Node,'FAX',R.Fax);
        -- add in children ...
        Loop
                Child := Generate_FacStaff_Dir.Get_Univ_Tel_Children(R.Orgn);
                exit when Child.Name is null;
                Gen_Entry(This_Node, Child);
        end Loop;
```

Figure 14: Directory generation Gen_Entry procedure.

Storing from a URL

While we are often able to generate our XML data files from data stored in the database, we sometimes need to load in other files (such as XSL template files) from other places. Another interface provided by Oracle, is the ability to take a URL, query the web, parse the XML document and return DOM format XML document. We wrapped that in our own routine (Figure 15) that then stores the resulting document in the document cache. Since we save the URL used in the document cache, we also have a Refresh_Url routine that will look up the URL from the document cache, repeat the query, and if the resulting document is valid, save the updated version in the cache.

```
function Cache_Url(
      url in varchar2,
      Schema_name in varchar2,
      Schema_instance in varchar2,
      d_title in varchar2,
      d_code in varchar2,
      d_desc in varchar2,
      d_type in varchar2)
return number;  -- Document_Id
```

Figure 15: Read and cache XML from a URL.

```
function Transform_And_Cache(
      Schema_name in varchar2,
      Document_Instance in varchar2,
      Transform_Document in varchar2,
      d_title in varchar2,
      d_code in varchar2,
      d_desc in varchar2,
      d_type in varchar2)
return number;  -- Document_Id
```

Figure 16: Transform and cache XML.

Unlike some of the other XML_Cache_Maint routines, this particular routine is generally only called from our web interface to the document cache. Since the URL query might fail, either due to connectivity problems[5] or the file obtained via the URL might no longer be a valid XML document. Longer term, this facility has much potential for transferring data into the database from foreign systems with the proper detection and reporting of failures.

Translation and Storing

In order to take advantage of the use of XML to separate data extraction from the presentation, and still take advantage of the version control support provided by the original Generate_File program, we need to be able to apply an XSL transform to an XML document within the database, and store the resulting document in the document cache. To this end, we have again wrapped the Oracle routines, with a function of our own (see Figure 16) to handle the translation and store the results back in the document cache.

[5]When Oracle processes this request, the database server attempts to make an HTTP connection to the specified web server.

With this version of the function, we supply the schema name, which is shared by both the Document Instance (the base XML document) and the Transform Document(the XSL transform document, which is just another XML document.) We also provide the document code for those instances with multiple documents. There is a version planned that will iterate over all of the sub documents in an instance, and regenerate all that are out of date (version control) automatically.

Conclusions and Futures

The ability to separate data extraction (like for the online directory) from the presentation has really helped in maintaining a consistent look and feel for the University web pages; prior to this work, the online directory pages lagged three to six months behind the rest of the web sites in their appearance. We still have a number of places where we generate HTML for both static pages, and for dynamic applications. Many of these may get rewritten to generate an intermediate XML format before the final HTML presentation.

Another area of exploration will be with DocBook [12], which is a markup standard markup for computer documentation and technical books developed by the joint efforts of Hal Computer Systems International, Ltd., and O'Reilly and Associates, Inc. This would be a good addition to the *Service Trak* [5] project to allow us to integrate service documentation with the current state of services at our site. XML will also dovetail nicely into Service Trak for the generation of configuration files for Big Brother and other service monitoring packages.

As use of the XML file generation grows, I expect we will be tuning and expanding the access control options for every stage of the document generation process. I also expect to make some refinements in the way we set up new targets for the Generate_File program. At this point, each new target requires a new PL/SQL package. With the Cache_Query support, a number of file generation targets can be reduced to a simple query and a few parameters. This will make it much easier to extract data from the database for external applications.

References and Availability

This project is part of (but not dependent on) the Simon system, an Oracle based system used to assist in the management of our computer accounts [7], enterprise white pages [4], printing configuration [3], All source code for the Simon system, including Generate File, is available on the web. See http://www.rpi.edu/campus/rpi/simon/README.simon for details. At present we have both AIX and Solaris versions of the Generate_File program in production and efforts are underway to finish a Java version. In addition, all of the Oracle table definitions as well as PL/SQL package source are available at http://www.rpi.edu/campus/rpi/simon/misc/Tables/simon.Index.html .

Some of the source code examples in this paper have been modified from the actual production code. I would suggest that if you are interested in this work, that you review the current source code available from the above URL.

Other Environments

XML has sparked a lot of interest from many people and as a result, there are a lot of XML and related resources available. I have found the SGML module for Emacs handy and there are many other editing tools that understand XML and XSL file formats. Along with editing environments, there are a number of programming interfaces and packages available, many of them for free. Aside from the PL/SQL packages provided by Oracle, there are Java, C++, perl and others.

Acknowledgements

I would like to thank Mario Obejas for his shepherding of this paper, as well as Jeff R. Allen, the LISA copy editor. I also want to thank Rob Kolstad for his excellent (as usual) job of typesetting this paper. Thanks also to Arlen Johnson and Kevin Bishop of Communication and Collaboration Technologies at Rensselaer (these are those wacky web guys who actually make good looking web pages) for their help with the XSL and XML design, and Alan, Andy, Bick and Kelly who helped review this paper.

Author Biography

Jon Finke graduated from Rensselaer in 1983 with a BS-ECSE. After stints doing communications programming for PCs and later general networking development on the mainframe, he then inherited the Simon project, which has been his primary focus for the past 12 years. He is currently a Senior Systems Programmer in the Networking and Telecommunications department at Rensselaer, where he continues integrating Simon with the rest of the Institute information systems. When not playing with computers, you can often find him building or renovating houses for Habitat for Humanity, as well as merging a pair of adjacent row houses into one. Reach him via USMail at RPI; VCC 319; 110 8th St; Troy, NY 12180-3590. Reach him electronically at finkej@rpi.edu. Find out more via http://www.rpi.edu/~finkej.

References

[1] Adler, Sharon, Anders Berglund, Jeff Caruso, Stephen Deach, Tony Graham, Tony Graham, Eduardo Gutentag, Eduardo Gutentag, Scott Parnell, Scott Parnell, and Steve Zilles, *Extensible Stylesheet Language (xsl) Version 1.0*, http://www.w3.org/TR/xsl, 2001.

[2] Bosak, Jon, *Extensible Markup Language (xml) version 1*, http://www.w3.org/TR/REC-xml, 1997.

[3] Finke, Jon, "Automating Printing Configuration," *USENIX Systems Administration (LISA VIII) Conference Proceedings*, pp. 175-184, USENIX, San Diego, CA, September, 1994.

[4] Finke, Jon, "Institute White Pages as a System Administration Problem," *The Tenth Systems Administration Conference (LISA 96) Proceedings*, pp. 233-240, USENIX, Chicago, IL, October 1996.

[5] Finke, Jon, "Automation of Site Configuration Management," *The Eleventh Systems Administration Conference (LISA 97) Proceedings*, pp. 155-168, USENIX, San Diego, CA, October, 1997.

[6] Finke, Jon, "An Improved Approach to Generating Configuration Files from a Database," *The Fourteenth Systems Administration Conference (LISA 2000)*, pp. 29-38, USENIX, New Orleans, LA, December, 2000.

[7] Finke, Jon, "Embracing and Extending Windows 2000," *The Sixteenth Systems Administration Conference (LISA 2002)*, USENIX, November, 2002.

[8] Finke, Jon, "Process Monitor: Detecting Events That Didn't Happen," *The Sixteenth Systems Administration Conference (LISA 2002)*, pp. 145-153, USENIX, November, 2002.

[9] Goossens, Michal and Sebastian Rahtz, *The LaTeX Web Companion*, Chapter 6, "Tools and Techniques for Computer Typsetting," Addison-Wesley, May 1999.

[10] Higgins, Shelly, *Oracle8i Application Developer's Guide – XML*, September, 2000.

[11] Portfolio, Tom, *PL/SQL Release 8 User's Guide and Reference*. Oracle Corporation, Part Num. A58236-01, December, 1997.

[12] Walsh, Norman and Leonard Muellner, *DocBook: The Definitive Guide*, O'Reilly and Associates, ISBN 1-56592-580-7, Sebastopol, CA, October, 1999.

Preventing Wheel Reinvention: The psgconf System Configuration Framework

Mark D. Roth – University of Illinois at Urbana-Champaign

ABSTRACT

Most existing Unix system configuration tools are designed monolithically. Each tool stores configuration data in its own way, has its own mechanism for enforcing policy, has a fixed repertoire of actions that can be performed to modify the system, and provides a specific strategy for configuration management. As a result, most tools are useful only in environments that very closely match the environment for which the tool was designed. This inflexibility results in a great deal of duplication of effort in the system administration community.

In this paper, I present a new architecture for system configuration tools using a modular design. I explain how this architecture allows a single tool to use different strategies for configuration management, enforce different ideas of policy, and prevent duplication of effort. I also describe the implementation of this architecture at my site and identify some areas for future research.

Introduction

Most Unix system configuration tools are composed of several common types of components:

- **Data Store**: The data store is the repository from which the tool reads the configuration data explicitly supplied by the system administrator. For example, ISConf [1] uses Makefiles, LCFG [2] uses X resources-style source files, cfengine [3] uses its own custom configuration file format, TemplateTree II [4] uses template files that are used to generate a cfengine configuration file, Arusha [5] uses XML files, and Simon [6] stores configuration data in an SQL database.
- **Policy Rules**: Policy is the programmatic manipulation of configuration data. Unlike the explicit configuration parameters read from the data store, policy rules act implicitly to enforce requirements imposed by the administrator. For example, at some particular site, "enable anonymous FTP" might automatically mean to install wu-ftpd, create the ftp user, add the appropriate entry to inetd.conf, update TCP wrappers, etc.
- **Actions**: Actions are the mechanisms for manipulating the system's state. For example, cfengine supplies specific file-editing actions like AppendIfNoSuchLine, while almost all existing tools allow you to execute arbitrary shell commands.

Despite these commonalities, most tools are very different from one another in the way that they put their components together to form a general strategy for configuration management. For example, ISConf takes the extreme approach of stepping through each state in a machine's configuration history in order to get to the current/desired state; cfengine is designed to converge from what a given file looks like now to what you want it to look like; and Simon is designed to generate individual configuration files from scratch.

Unfortunately, most existing tools are designed monolithically, which means that you cannot choose the components or strategy independently. For example, there's no way to use cfengine's convergence strategy with Simon's SQL data store. There's also no way to use Simon's file generation strategy for some configuration files and cfengine's convergence strategy for others.

This inflexibility results in a great deal of duplication of effort in the system administration community, because the only way to change out a single component or add support for a different strategy is to write a whole new tool. The new tool may include new components or support a different strategy, but a great deal of time is also spent duplicating existing components from other tools.

A New Approach: The psgconf System

Here at the University of Illinois at Urbana-Champaign (UIUC), I've developed a tool called psgconf to address these problems. Instead of being a monolithic tool with a fixed set of components, psgconf dynamically loads external modules that implement its components. This makes it very easy to add or remove components as needed.

The psgconf system is written entirely in Perl, and its components are nothing more than Perl objects that provide the appropriate methods for the component type. This means that writing a new component for psgconf is as simple as writing a Perl module that provides the component's object class.

Object Structure

Each of the next subsections describes an object that is part of the psgconf system.

The PSGConf Object

The central object in the psgconf system is provided by the PSGConf module. The PSGConf object is extremely simple; its primary function is to coordinate the activity of the other objects, which are where most of the work is actually done.

Data Objects

Instead of storing configuration data in static variables, psgconf encapsulates data in Data objects. This provides a great deal of flexibility in configuring the system, because each Data object class can provide whatever methods are appropriate for the encapsulated data.

All Data object classes should be derived from the PSGConf::Data base class. The base class provides several fundamental methods, such as set(), get(), and unset(). However, new subclasses are free to override these methods or to add whatever new methods are appropriate for the type of data the class intends to represent. The psgconf system includes several such modules that provide useful Data object classes:

Data Object Class	Description
PSGConf::Data::Boolean	boolean data
PSGConf::Data::Hash	hash table data
PSGConf::Data::Integer	integer data
PSGConf::Data::List	list data
PSGConf::Data::String	character string data
PSGConf::Data::Table	table-oriented data

For example, the PSGConf::Data::String module provides a Data object class for character-string data. It provides several methods that are specially tailored for manipulating string data: append(), which appends its argument to the end of the string already contained by the object; prepend(), which prepends its argument to the beginning of the string already contained by the object; and gsub(), which replaces substrings in the string already contained by the object.

As each Data object is instantiated, it is registered with the central PSGConf object under a particular name, which can then be used to reference the object. For example, there might be a PSGConf::Data::String object called log_dir or a PSGConf::Data::Boolean object called anon_ftp_enable.

DataStore Objects

In the psgconf framework, a system's configuration is expressed as a series of configuration statements that manipulate the registered Data objects. Each statement consists of the name of a Data object, a method to call on that object, and a list of arguments to pass to that method. The mechanism for reading these configuration statements is provided by DataStore objects.

Each DataStore object provides a method called read_config() that reads configuration statements from some arbitrary source and executes the requested method calls. It might read the statements from a local configuration file using some particular syntax, or it might use a network protocol to read the statements from a remote configuration server (referred to as a "gold server"). It does not matter to psgconf how the statements are actually read, as long as the DataStore module can read the statements and execute the requested method calls.

For example, the PSGConf::DataStore::ConfigFile DataStore object reads configuration statements from a local file using syntax that looks like this:

```
log_dir->set "/var/log";
```

This statement tells the DataStore object to look for a Data object named log_dir and call that object's set() method with the string /var/log as its argument.

Action Objects

Action objects represent actions to be performed on the system.

All Action object classes should be derived from the PSGConf::Action base class, which provides several support methods. However, new subclasses should always define three important methods: check(), which determines whether the action actually needs to be performed or whether the system is already in compliance; diff(), which displays details about the changes that would be made to the system; and do(), which actually performs the action.

The psgconf system includes many useful Action classes. See Table 1 for several examples. Action objects are processed in the order in which they are registered. This is discussed in more detail below.

Control Objects

Control objects are the real workhorses of the psgconf system. They are responsible for directing the overall process of configuring the system.

From their constructor, Control objects can instantiate and register new Data objects. As mentioned above, each Data object is given a name as it is

Action Object Class	Description
PSGConf::Action::GenerateFile	programmatically generate a file
PSGConf::Action::MkDir	create a directory
PSGConf::Action::ModifyFile	programmatically modify an existing file
PSGConf::Action::RunCommand	run an external command
PSGConf::Action::Symlink	create a symbolic link
PSGConf::Action::TouchFile	create an empty file

Table 1: Examples of Action classes.

registered. For example, the PSGConf::Control::AnonFTP module registers the Data objects shown in Table 2.

Control objects can also provide any number of policy methods, which perform programmatic manipulation of Data objects. As with Data objects, the Control module's constructor registers each policy method under a particular name. Going back to the previous example, the PSGConf::Control::AnonFTP module might register the policy methods in Table 3. Note that just because a policy method is registered does not necessarily mean that it will be used. This is discussed in more detail below.

Control objects also provide a method called decide(), which is responsible for instantiating and registering new Action objects based on the final values of the Data objects. For example, the PSGConf::Control::inetd module's decide() method checks whether the inetd Data object contains any entries, and if so it instantiates a PSGConf::Action::GenerateFile object to create the inetd.conf file.

The Big Picture

To illustrate how all of these object types fit together, it is useful to enumerate the steps taken by psgconf to configure the system.

Step 1: Control and DataStore Module Instantiation

The PSGConf object starts by reading the /etc/psgconf_modules file, which can contain the following types of entries:

```
DataStore module_name [args ...]
Control   module_name [args ...]
Policy    method_name
```

The DataStore entries specify a DataStore module to be instantiated. Any additional arguments are passed to the module's constructor. Multiple DataStore entries can be present, in which case the modules are accessed in the order in which their entries appear in the /etc/psgconf_modules file.

The Control entries specify a Control module to be instantiated. As with DataStore entries, any additional arguments are passed to the module's constructor. Multiple Control entries can be (and usually are) present, in which case the modules are accessed in the order in which their entries appear in the /etc/psgconf_modules file. As mentioned above, the constructor for each Control object can instantiate and register any needed Data objects, as well as registering any policy methods it provides.

The Policy entries specify the set of policy methods that will actually be invoked. The methods will be

Data Object	Class	Description
anon_ftp_enable	PSGConf::Data::Boolean	whether or not to enable anonymous FTP
anon_ftp_dir	PSGConf::Data::String	path to ftp root
anon_ftp_options	PSGConf::Data::Hash	options for ftpaccess file

Table 2: Data objects registered by the PSGConf::Control::AnonFTP module.

Policy Method	Description
anon_ftp_add_user	if anon_ftp_enable is set, create ftp user
anon_ftp_add_inetd_entry	if anon_ftp_enable is set, add inetd.conf entry
anon_ftp_add_package	if anon_ftp_enable is set, install wu-ftpd

Table 3: Policy methods registered by the PSGConf::Control::AnonFTP module.

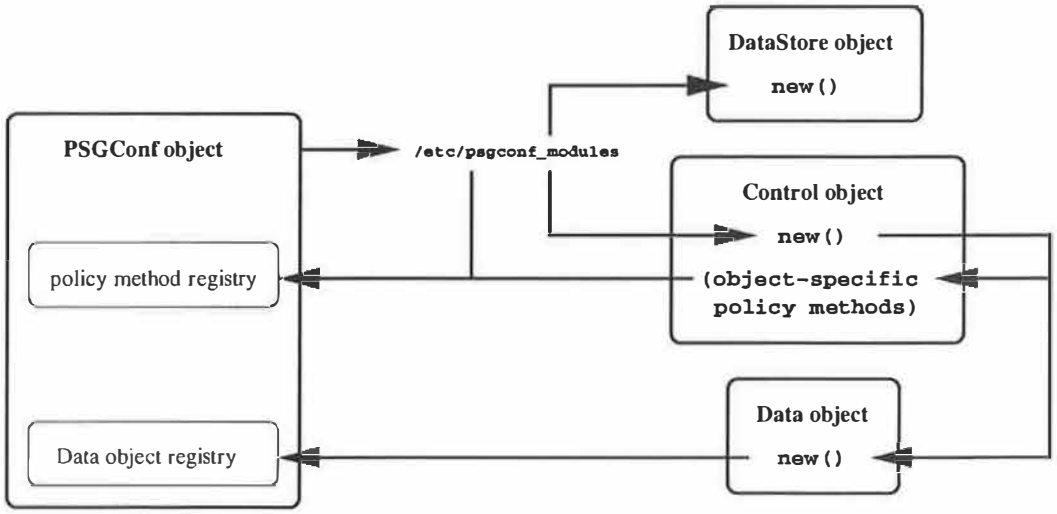


Figure 1: Control and DataStore module instantiation.

invoked in the order in which their entries appear in the /etc/psgconf_modules file. Note that as psgconf reads each Policy entry, it will verify that the specified policy method is actually registered. This means that it is an error to specify a Policy entry for a policy method before the Control entry for the Control module that provides that policy method. By convention, this problem should be avoided by placing all Policy entries at the end of the file.

Step 2: DataStore Processing

The PSGConf object loops through the list of DataStore objects specified in the /etc/psgconf_modules file and calls the read_config() method of each object. The read_config() method accesses the data store and reads configuration statements. Each configuration statement results in calling a method of one of the Data objects that was previously registered by the Control modules.

Step 3: Policy Enforcement

The PSGConf object calls each of the policy methods specified in the /etc/psgconf_modules file. The policy methods programmatically manipulate Data objects to enforce policy.

Step 4: Action Instantiation

The PSGConf object loops through the list of Control objects specified in the /etc/psgconf_modules file and calls the decide() method of each object. The decide() method instantiates Action objects to perform the appropriate actions based on the content of the Data objects.

Note that the Action objects are processed in the order in which they are registered. Because they are registered by a Control object's decide() method, this means that the order of the Control objects in the /etc/psgconf_modules file actually determines the order of the Action objects instantiated by the various Control modules.

Step 5: Action Checking

The PSGConf object loops through the list of Action objects and calls the check() method of each object. The check() method checks to see if the action actually needs to be performed.

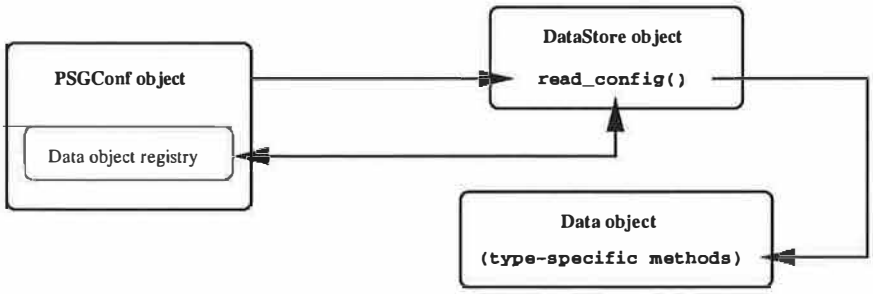


Figure 2: DataStore processing.

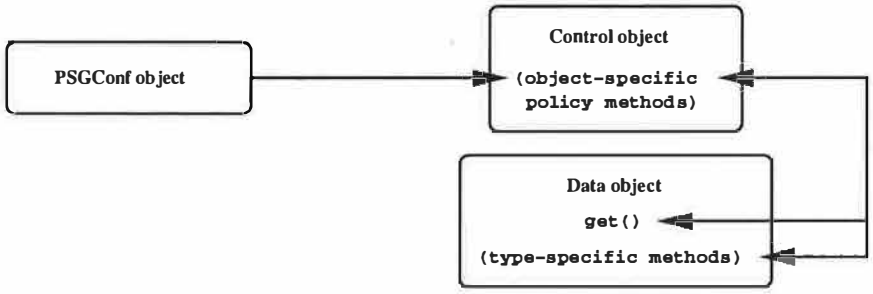


Figure 3: Policy enforcement.

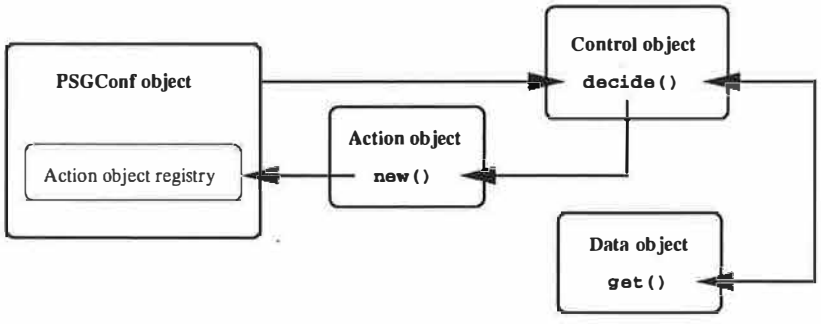


Figure 4: Action instantiation.

For example, the PSGConf::Action::GenerateFile module's check() method will generate the new file and compare it to the existing file; if the two files differ, then the action needs to be performed.

Step 6: Action Implementation

For each Action object that needs to be performed, the PSGConf object calls the object's diff() and/or do() method, depending on what options psgconf was invoked with.

The Action object's diff() method prints the details of the change that will be made to the system. For example, the PSGConf::Action::GenerateFile module's diff() method invokes the diff(1) command to show the differences between the existing file and the new file, which was generated by the check() method in the previous step.

The Action object's do() method actually performs the change on the system. For example, the PSGConf::Action::GenerateFile module's do() method replaces the existing file with the newly generated version.

Step 7: Cleanup

The PSGConf object loops through the list of Control objects and calls the cleanup() method of each object, if present. The cleanup() method performs any necessary cleanup tasks, such as restarting a daemon after its configuration file has changed.

The Dual Role of Control Modules

In practice, Control modules are written to serve one of two distinct roles: they can encapsulate the configuration of a particular subsystem, or they can provide features that span multiple subsystems.

Most Control modules are designed to encapsulate the configuration of a specific subsystem. For example, the PSGConf::Control::AnonFTP module provides all of the necessary Action objects for creating the anonymous FTP tree, generating the /etc/ftpaccess file, and so on. Essentially, it contains everything necessary for configuring wu-ftpd.

Control modules can also be designed to provide subsystem-independent features. For example, my group uses a locally written Control module called PSG::Control::ConnectionLog that configures TCP wrappers to log connection information via syslog to a file called connections. It does this via a policy method that manipulates Data objects provided by the PSGConf::Control::syslog and PSGConf::Control::TCPWrappers modules. However, the PSG::Control::ConnectionLog module does not register any Action objects of its own, because it does not configure any subsystems directly.

Although these two roles are often distinct, they are both provided by the same type of module (Control modules) because there is a great deal of overlap in the mechanisms they use. Both may need to provide Data objects to make the relevant subsystem or feature configurable, and both may need to provide policy methods to manipulate Data objects provided by other modules. Rather than duplicate all of this functionality in two different module APIs, psgconf is able to handle both roles through the same interface.

Ordering Policy Methods

The order in which policy methods are invoked is important, because a given policy method may depend on changes made by another policy method.

For example, the PSGConf::Control::AnonFTP module provides a policy method called anon_ftp_add_tcpd that adds an entry to the tcp_wrappers Data object for the FTP server. In addition, my group's locally written PSG::Control::ConnectionLog module (described above)

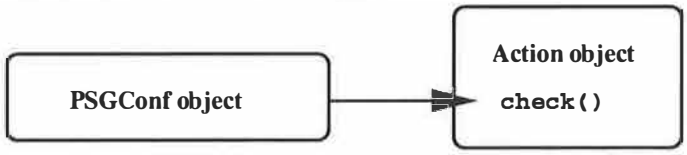


Figure 5: Action checking.

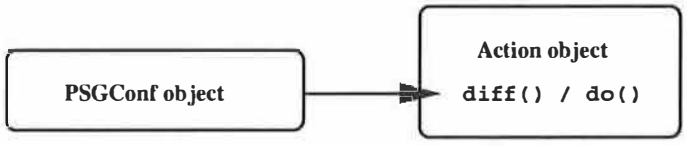


Figure 6: Action implementation.

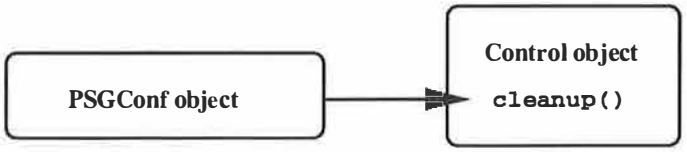


Figure 7: Cleanup.

provides a policy method called connection_log_modify_tcpd. This policy method checks the syslog Data object (provided by PSGConf::Control::syslog) for a logfile called connections. If found, it adds a severity option to each entry in the tcp_wrappers Data object so that wrapper information is sent to the connections log.

When the connection_log_modify_tcpd method is invoked, it adds the severity option for each entry that it finds. However, if the entry for the FTP server is not added until after the connection_log_modify_tcpd method runs, then the severity option will not be added to that entry. Therefore, the anon_ftp_add_tcpd policy method must be invoked before the connection_log_modify_tcpd policy method.

At first, I considered implementing a mechanism that would automatically determine the order in which to invoke the policy methods based on dependency information encoded into the Control module by the module's author. However, the module's author doesn't have any way of knowing what other modules or policy methods the module will be used with, so there's no way to encode all of the dependencies into the module. Only the administrator can do that, because he's the one choosing the Control modules and policy methods.

As a result, psgconf allows the administrator to set the order in which policy methods are invoked in the /etc/psgconf_modules file.

A Note About Object-Orientation

Note that I chose to use object-oriented modules for three main reasons. First, that's the established convention for writing new Perl modules. Second, the object-oriented approach provides an easy way for each object to maintain its own state. This is especially important for Data and Action objects, because multiple independent objects are often instantiated from each class. Finally, the object-oriented approach makes it possible to create new modules as subclasses of existing modules, which can make it easier to write and maintain variations of existing modules.

Advantages of psgconf

The psgconf system is extremely flexible. Several of its more noteworthy advantages are described here.

Flexible DataStore Mechanism

Because psgconf is not tied to a single DataStore implementation, it can grow with its environment as the environment's needs change. For example, the only existing DataStore implementation is the PSGConf::DataStore::ConfigFile module described above, which reads configuration data from a local configuration file. In the future, I plan to develop new data store modules to access configuration data from a central gold server using mechanisms like SQL queries or XML-RPC calls. Using these new data store modules will be as simple as updating the /etc/psgconf_modules file.

A Smorgasbord of Strategy

Because new Action classes can be created as needed, psgconf is not limited to a specific strategy for configuration management. For example, cfengine's convergence strategy can be implemented for certain files using objects of the PSGConf::File::ModifyFile class, while other files can be generated from scratch using PSGConf::File::GenerateFile objects. Because Action objects are processed in a specific order, even ISConf's history-recreation strategy can be implemented, simply by adding a new Control module for each state in the configuration history.

Separation of Config Data from File Formats

Because configuration data is represented in Data objects and specific file formats are understood by particular Action and Control objects, there is a clean separation of code and data. This makes it very easy to support platforms that each use a different file format to represent the same data. For example, the PSGConf::Control::PAM module provides a single Data object to contain all of the PAM configuration data, but it can instantiate Action objects to generate either /etc/pam.conf or individual files in /etc/pam.d, as appropriate. Similarly, a module might instantiate Action objects to generate either inetd.conf or xinetd configuration files based on the same Data objects.

Future Directions

There is much potential for improvement in the area of communication between package management tools and configuration management tools. Currently, using a psgconf Control module for something like Apache may require the administrator to override a lot of defaults, because the module does not know what default paths or features were built into the Apache package at compile time. If this information were encoded into the Apache package in some standard way, the psgconf module could query the package manager to get this information.

There is currently no mechanism for asynchronously notifying psgconf of changes in a central data store. As new types of DataStore modules are developed, this functionality will be implemented.

Although psgconf is publicly available, it has never been announced in any major public forums, and I am not aware of any other large sites using it. I would very much like to get feedback from others who are using it, so that any lingering features or assumptions that might be specific to my site can be eliminated.

Ultimately, I would like to create a CPAN-style site for distributing psgconf modules written by different people. The goal of this site would be to allow system administrators in different organizations to share psgconf modules, thus avoiding the need to reimplement a module that's already been written.

Conclusion

The psgconf framework's modular architecture provides a great deal of flexibility. Components can be swapped out to meet changing needs, new platforms can be supported without changing the configuration data model, and different strategies can be combined to manage system configuration in the most flexible manner. Eventually, I hope that the ability to share modules will help prevent wheel reinvention in the entire system administration community.

Availability

The psgconf package can be downloaded from its home site at http://www-dev.cites.uiuc.edu/psgconf/. It is distributed under a BSD-style license.

Acknowledgments

I would like to thank Paul Anderson, Alva Couch, Daniel Hagerty, Luke Kanies, Adam Moskowitz, and many others for providing frequent sanity checks as I was developing the psgconf framework.

I would also like to thank Steve Traugott for helping the system administration community think about infrastructure in a whole new way.

Author Information

Mark Roth is the technical lead of the Production Systems Group of the Campus Information Technologies and Educational Services department at the University of Illinois at Urbana-Champaign. Mark is the author of several open-source software packages. He can be contacted via email at roth@uiuc.edu, and his web page is http://www.uiuc.edu/ph/www/roth.

References

[1] Traugott, Steve and Joel Huddleston, "Bootstrapping an Infrastructure," *LISA XII Proceedings*, Boston, MA, pp. 181-196, 1998.

[2] Anderson, Paul, "Towards a High-Level Machine Configuration System," *LISA VIII*, Berkeley, CA, pp. 19-26, 1994.

[3] Burgess, Mark, "Cfengine: A Site Configuration Engine," *USENIX Computing Systems*, Vol. 8, Num. 3, http://www.cfengine.org/, 1995.

[4] Oetiker, Tobias, "TemplateTree II: The Post-Installation Setup Tool," *LISA XV Proceedingsk*, San Diego, CA, pp. 179-186, 2001.

[5] Holgate, Matt and Will Partain, "The Arusha Project: A Framework for Collaborative Unix System Administration," *LISA XV Proceedings*, San Diego, CA, pp. 187-198, 2001.

[6] Finke, Jon, "An Improved Approach for Generating Configuration Files from a Database," *LISA XIV*, New Orleans, LA, pp. 29-38, 2000.

SmartFrog meets LCFG: Autonomous Reconfiguration with Central Policy Control

Paul Anderson – University of Edinburgh
Patrick Goldsack – HP Research Laboratories
Jim Paterson – University of Edinburgh

ABSTRACT

Typical large infrastructures are currently configured from the information in a central configuration repository. As infrastructures get larger and more complex, some degree of autonomous reconfiguration is essential, so that certain configuration changes can be made without the overhead of feeding the changes back via the central repository. However, it must be possible to dictate a central policy for these autonomous changes, and to use different mechanisms for different aspects of the configuration.

This paper describes a framework which can be used to configure different aspects of a system using different methods, including explicit configuration, service location, and various other autonomous techniques. The proven LCFG tool is used for explicit configuration and to provide a wide range of configuration components. The dynamic elements are provided by the SmartFrog framework.

Introduction

Typical large infrastructures (see [7] for some case studies) are currently configured from the information in a central configuration repository, such as sets of hand-crafted cfengine [11] scripts, or LCFG [9] source files. Changes to individual nodes are made by editing these central descriptions and running the appropriate tool to reconfigure the node.

As infrastructures get larger and more complex, some degree of autonomous reconfiguration is essential, so that individual nodes (and clusters) can make small adjustments to their configuration in response to their environment, without the overhead of feeding changes back via the central repository. For example, the members of a cluster might elect a replacement for a failed server amongst themselves, without requiring a change to the central configuration server; see the example in Figure 1.

Peer-to-peer style autonomous reconfiguration is already present in several tools, such as ZeroConf [17] (as used by Apple's Rendezvous) and other systems using Service Location Protocols (for example, [19]). However, a completely autonomous approach is not suitable for large installations; there needs to be some central control over the policy under which the autonomous choices are made; for example, exactly which nodes are eligible to be elected as a replacement server? There will also be a good deal of configuration information that does need to be specified explicitly, and there may need to be several different simultaneous techniques for making autonomous decisions.

LCFG [9] is a proven, practical tool for centralized configuration management of large, diverse infrastructures. SmartFrog [16] is a flexible, object-oriented framework for deployment and configuration of remote Java objects. This paper describes the architecture of a combined LCFG/SmartFrog framework which uses LCFG to install a complete system (including SmartFrog) from scratch. SmartFrog components on the resulting system are then able to take control of arbitrary LCFG components and configure them autonomously, according to policies defined in the central LCFG configuration database.

This approach allows the configuration of various aspects of the system to be shifted easily between explicit

Explicit Specification:

- Node X must run a print server
- Node Y must run a print server

Policy Specifications with autonomous configuration:

- Any Linux server may run a print server
- Nodes A,B,C,V,X,Y,Z are servers
- Nodes P,Q,R,X,Y,Z are Linux machines
- There must be exactly two print servers

The nodes can decide "among themselves" which of the eligible nodes actually run as print servers. If one of them fails, they can re-elect a replacement without a change to the central policy.

Figure 1: Explicit specification vs policy specification – an example.

central specification, and autonomous control, using one of several different procedures for making the autonomous decisions, and performing the related peer-to-peer communication. No change is required to the software components that actually implement the configuration.

The combined framework makes an ideal test bed for experimenting with different models of autonomous configuration in a real environment. This paper describes the testbed, together with some demonstrator applications, including a complete Grid-enabled (OGSA [15]) printing service, with dynamic server re-allocation and failure recovery.

The next section describes the background in more detail, including an overview of the LCFG and SmartFrog tools, and the motivation towards more dynamic reconfiguration. Subsequent sections describe the combined LCFG/SmartFrog framework, present some simple example applications, and explicate the OGSA print service demonstrator.

Background

LCFG

LCFG is an established configuration framework for managing large numbers of Unix workstations. Originally developed under Solaris (see [4]) using NIS to transport configuration parameters, the current version (see [9]) runs under Linux and uses XML/HTTP for parameter transport. LCFG acts as an evolving testbed for configuration research, as well as a production system for the infrastructure in the School of Informatics at Edinburgh University. An older version is also in use on the testbeds for the European DataGrid Project [1]. LCFG includes around 70 modules for managing a wide range of different subsystems, ranging from PCMCIA configuration on laptops, to OGSA web services for Grid farms (see [5]).

Figure 2 shows the overall architecture of the LCFG system:

- The configuration of the entire site is described in a set of declarative *LCFG sources files*, held on a master server. These source files are managed by many different people and describe various different *aspects* of the overall configuration, such as "a web server" or a "laptop", or a "student machine".
- The *LCFG compiler* monitors changes to the sources files and recompiles the source for any nodes whose aspects have changed.
- The result of the compilation is one XML *profile* for each node. The profile defines explicitly all the configuration parameters (called *resources*) for a particular node. This includes the list of software packages, as well as the values to be used in all the various configuration files. Given a repository of software packages, the information in the profile is sufficient to completely reconstruct the node from scratch; LCFG can install a new machine from the bare metal, or clone existing ones, using just a profile, and an RPM repository.
- Client nodes receive a simple notification when their profile has changed,[1] and they collect their new profile using HTTP(S) from a standard web server.
- The client includes a set of *component* scripts, each responsible for a self-contained subsystem, such as inetd, or kdm. Components are

[1]Clients also poll the server in case the notification is lost.

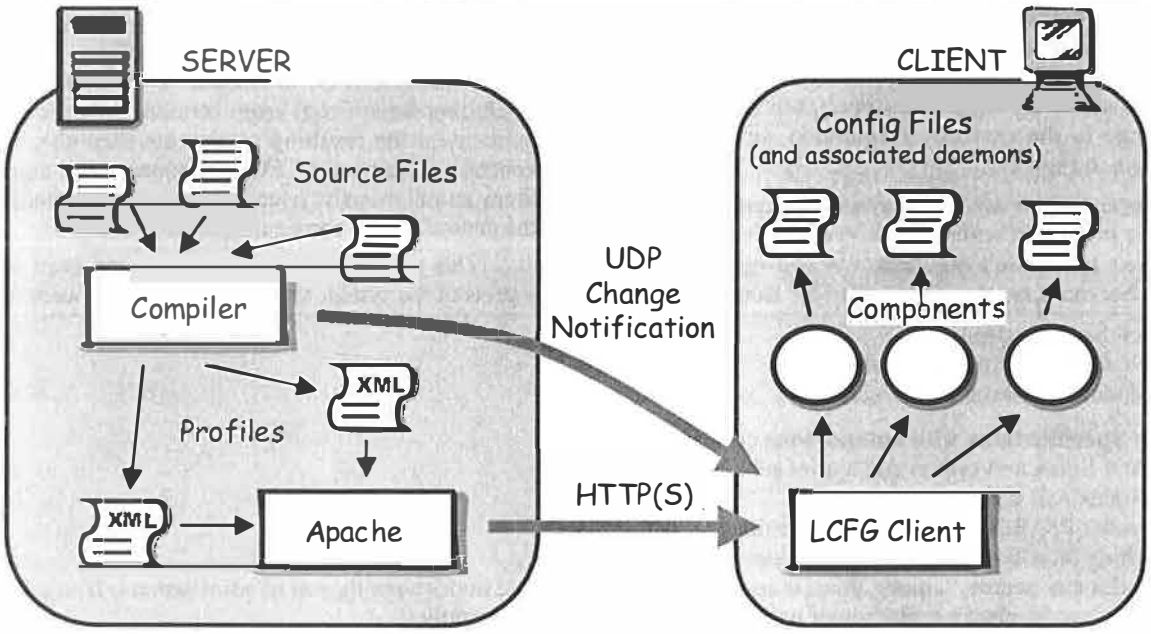


Figure 2: The LCFG architecture.

called whenever any of their resources are changed, and they are responsible for recreating the configuration files that they manage, and taking any other necessary action, such as restarting daemons. Note that each configuration file is managed only by a single component, avoiding any problems with conflicting changes and ordering of updates.

- The LCFG implementation also contains a simple monitoring mechanism (not shown in the diagram) that returns component status information to the server for display on a status page.

Note that there is no one-to-one correspondence between the source files and the profiles, nor between the profiles and the components; source files represent logical aspects of the site configuration,[2] profiles represent individual node configurations, and components manage particular subsystems of a host.

The architecture of LCFG has many well-recognized advantages, which are described more fully in the references, and the basic principles have been adopted for other systems such as [12]. In particular, the single source of configuration information allows a complete site to be reconstructed from scratch. This means that the complete configuration information is always available, and configurations can be validated before deployment by checking the source files.

However, certain configuration information may only be available on the client node; for example, a roaming laptop might need to define a new network configuration while disconnected from the main server; or some information may be obtained from a dynamic source such as DHCP, or DNS SRV records. LCFG includes a mechanism known as *contexts* for handling simple inclusion of some configuration parameters from other sources. This is sufficient to support the above examples, but inadequate for more extensive dynamic reconfiguration.

SmartFrog

SmartFrog is a distributed service and resource configuration engine designed to install and manage complex services spread over a number of computing nodes and other resources. It has been designed to handle the need for distributed sequencing and synchronization of configuration actions, as well as coping with the complexities introduced by the dynamism inherent in such large distributed environments, such as those introduced by partial system failure and communication problems.

The SmartFrog system consists of a number of aspects:

- A declarative description notation for defining desired configuration states, service life-cycles and dependencies between the various service components, including the work-flows to carry out the required changes.

  The notation provides a number of features that make it specifically useful for its purpose. It allows the definition of complex structured data and dependencies, with validation predicates to provide rich correctness criteria for their use and modification. The data is defined through the use of templates, with flexible operators to combine and modify the templates to create the final desired form for instantiation.

  This process may be linked to database queries or to active run-time discovery services to provide templates with late-bound and dynamic configuration models.
- A component model defining how configurator components – those which carry out the various configuration and other management tasks on the resources and services – are to be implemented. These may then be deployed and managed by the SmartFrog systems as part of the configuration work-flows.

  A number of useful pre-defined components are provided as part of the management framework, to support aspects such as resource and service discovery, service failure detection, and script and command execution.
- A distributed deployment and management run-time environment which uses the descriptions and component definitions to orchestrate the work-flows to achieve and maintain the desired state.

  Unlike many configuration systems, the environment does not merely support a run-once installation model of configuration – it supports the use and description of persistent components that monitor service state and take appropriate corrective action to achieve continual service availability or other closed-loop control aspects such as ensuring service-level guarantees.

  In addition, there is no central point of control, no point to act as a bottle neck in the system. Work-flows are fully distributed and not driven by a central point. Hooks are available to track the progress of these work-flows, and to locate the various management components that are started as a consequence. Thus a rich set of tools may be developed that help in tracing, monitoring and managing the SmartFrog run-time system.

The SmartFrog system provides a security framework to ensure that all configuration actions are valid and authorized by an appropriate authority. The model supports a number of separate security domains, thus ensuring partitioning of responsibility and limiting accidental interaction between these domains. All configuration descriptions and configuration component code must be signed, and these signatures are checked at all points of the configuration process to ensure the integrity of the service configuration process.

[2] Although individual nodes also have node-specific source files

The SmartFrog system lacks a number of features that are necessary in a complete configuration system, and which are largely supplied by the integration with LCFG.

The first of these is that SmartFrog assumes that the underlying resources are already running, complete with their OS image. It provides no help in taking a node from bare metal to running system. SmartFrog starts from the assumption that a node is booted from one of a small set of minimal images at which point a SmartFrog system could configure the various services. LCFG provides the capability to carry out this bootstrap phase.

The second is that SmartFrog is not currently a complete solution; it is a framework for building such solutions. For example, it does not contain a repository for configuration descriptions, nor does it enforce any specific way in which configuration descriptions are to be triggered – these may be triggered by external entities (such as the LCFG system) or by configuration components executing within the SmartFrog framework.

Finally, the SmartFrog framework has yet to be provided with a large collection of service-specific configuration components – such as ones for configuring DNS, DHCP, printers and print queues, and so on. LCFG, however, has been developed over many years to provide precisely this collection of components. A good integration that allows SmartFrog components to wrap and use those of LCFG would provide the best of both worlds.

Dynamic Reconfiguration

There has recently been a growing recognition that computer systems need to support more *autonomic* reconfiguration (for example, [14]) if we are to build ever-larger and more complex systems with an acceptable level of reliability. This requires automatic reconfiguration, *not just of individual nodes*, but of the higher-level roles and interconnections between the nodes. For example, if a server goes down, it should be possible to reconfigure another node to take over this function, and to redirect all the clients to this new server. This is possible with the current LCFG; some automatic tool could simply make the appropriate changes to the LCFG source files, and the whole network could be restructured. However, this involves a large feedback loop via the central configuration server. This provides a single point of failure and a centralized architecture which is inherently unsuitable for very large-scale dynamic systems. We would like to see a much more distributed system where the central server could define a high level policy, and small clusters of nodes could agree, and change, the details of their configuration autonomously within the limits defined by the central policy.

In addition to the fault-tolerance example mentioned above, load-balancing is another case where we would like to make transient, autonomous configuration changes that do not really represent fundamental changes to the static configuration of the fabric; for example, we might want to stop and start additional web servers on a number of nodes to match the demand. The central configuration should define the set of eligible nodes, but we probably do not want to change the central configuration specification every time the load changes.

There is also one less obvious advantage in devolving the detailed configuration decisions to some autonomic agent; at present, users (sysadmins) are forced to specify explicit configuration parameters, when very often, they only need to specify a more general constraint; for example it might be necessary to specify "Node X runs a DHCP server," when all that is really required is "There should be one DHCP server somewhere on this network segment." This unnecessary explicitness means that the compiler is often unable to resolve conflicts between different aspects, and manual intervention is required; for example, when somebody else removes "Node X."

Previous Work

Reference [6] is a report from the GridWeaver project that includes a thorough survey of existing system configuration tools, together with an attempt to classify common features and different approaches. This report includes a comprehensive list of references to other system configuration tools which are not reproduced here. Very few of these tools even support a clear declarative description of the desired configuration state, and none provide the ability to specify high-level policy about the configuration of a fabric together with a mechanism to enforce it.

Most people will however, be familiar with a number of specific tools that do provide dynamic reconfiguration according to central policy; for example, DHCP dynamically configures IP network addresses within the range specified by the policy embedded in the server configuration. There is currently much interest in more dynamic configuration of network parameters, for example the IETF ZeroConf [17] working group aims to:

- Allocate addresses without a DHCP server.
- Translate between names and IP addresses without a DNS server.
- Find services, like printers, without a directory server.
- Allocate IP Multicast addresses without a MADCAP server.

However, large sites will almost certainly want to define the policy within which these autonomous tools operate.

At the opposite end of the scale, there has been some work on dynamic configuration of specific services, particularly web services. Poyner's paper [19] is a good example that describes the use of a Service Location Protocol (SLP) and a centrally defined policy to dynamically configure web services for load-balancing and fault-tolerance.

All the above examples are application-specific implementations, and we are not aware of any attempt to integrate a generic facility for autonomous reconfiguration into a general-purpose system configuration framework. The following sections describes how LCFG and SmartFrog have been combined to construct an experimental framework that does provide this ability to apply different policies and autonomous techniques to arbitrary aspects of a system configuration.

Combining SmartFrog and LCFG

The integration of LCFG and SmartFrog has been achieved in the following way (this is shown diagrammatically in Figure 3):

- The SmartFrog framework has been wrapped as an RPM that can be installed on any node using the existing LCFG software installation components.
- An LCFG component has been developed to manage the SmartFrog Daemon running on a node. This component has two main purposes:
  1. To control the life cycle of the SmartFrog Daemon (typically it will be started at boot time).
  2. To control the list of deployed SmartFrog components, and their configuration information. This allows the central LCFG repository to control the *general policy* under which the SmartFrog framework runs.
- Once deployed, a SmartFrog component can take responsibility for the management of any part of a fabric. However, SmartFrog does not have the extensive range of components that LCFG provides for managing the various aspects of system configuration (apache, fstab entries, etc.). A generic SmartFrog → LCFG adaptor component has been developed that allows the SmartFrog framework to interact and control any LCFG component. This adaptor allows a SmartFrog component to act as a proxy for an underlying LCFG component. This gives the SmartFrog framework access to all the configuration capabilities of LCFG.

The SmartFrog framework makes it easy for SmartFrog components to perform peer-to-peer interactions with each other. With the combined LCFG and SmartFrog framework these peer-to-peer interactions can lead to re-configurations of the base fabric set-up by the central LCFG server. There are various peer-to-peer mechanism built into SmartFrog:

- Service Location Protocols. These allow SmartFrog components to automatically discover each other without the need for any explicit configuration to link them.
- Partition Membership Protocols. These allow a number of SmartFrog components to declare themselves as part of a group and have the framework automatically elect a leader within the group. If the leader fails, a new leader will be elected in its place. This is typically used to provide fail-over for critical services such as DHCP (The DHCP example is discussed in more detail later).
- Java Remote Method Invocation (RMI). The SmartFrog framework provides look-up services that allow any component to locate any other component on the network by name. Java RMI then allows these components to interact, exchange information and make decisions about required fabric re-configurations.

See the two next sections for a number of examples illustrating the practical use of these concepts.

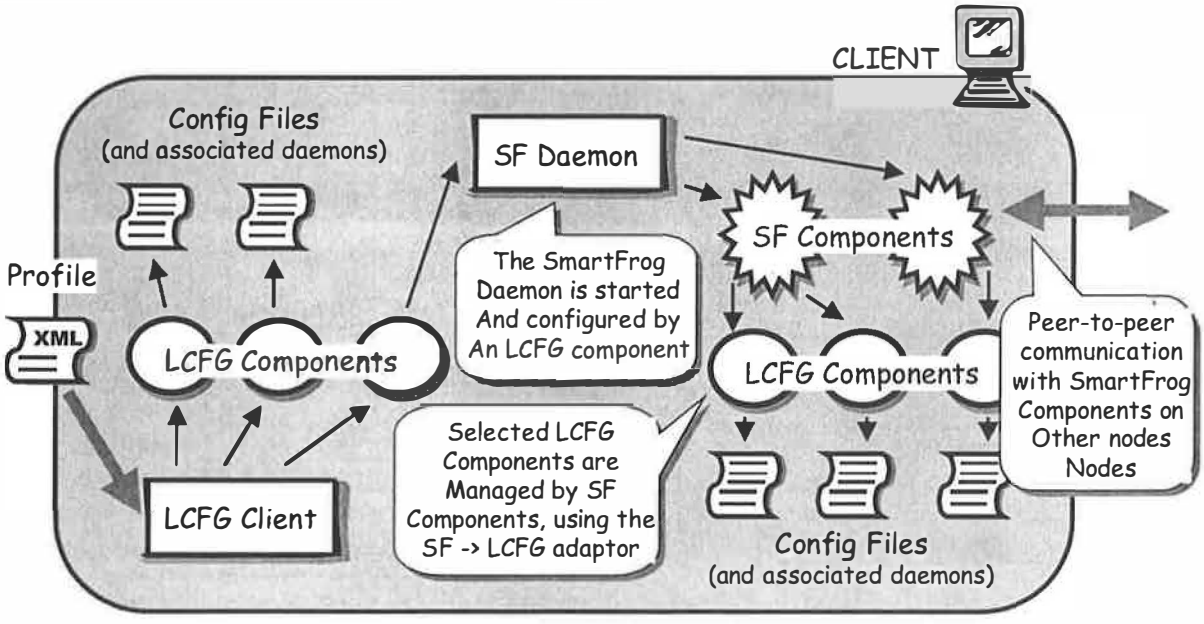


Figure 3: The SmartFrog/LCFG integration architecture.

Example Applications

The importance of the move from single node configuration to dynamic multi-node coordinated configuration can be illustrated by the use of a couple of examples. The two examples presented here consider different aspects of dynamic configuration: the first uses the underlying framework mechanisms to provide service reliability and failure recovery, the second examines the use of discovery for automatically and dynamically adjusting to service location changes.

Service Reliability

A scenario that frequently occurs is that of a service requiring a minimum number of daemons to exist on a collection of servers thereby ensuring a specific service reliability level. So, for example, it may be desirable for there to be at least two instances of a DHCP service on a specified collection of servers. This is relatively easily described: a configuration description would state which servers should hold an instance of the DHCP daemon. Descriptions of this kind would be validated to ensure that two are defined.

However server failures do occur, and it is necessary that the failure of a server containing such a daemon results in the automated re-deployment of the "spare" daemon onto another server thus maintaining the guaranteed service level.

The configuration problem can be described as follows: it would be best if the configuration description could be provided as a set of constraints regarding service replication over a collection of independent nodes, rather than a fixed static mapping of daemons to servers. These constraints should be maintained without needing to define a new static association.

Consider the following base configuration. Each server of a set of servers is configured with two components: a group membership component and a configuration policy engine.

A group membership component is one that uses a network protocol to decide which of a possible collection of such components (each representing their server) are healthy and able to run one or more of the daemons. This protocol must ensure that all servers that are part of this collection agree on its members. From this information a leader may easily be elected.

Such protocols are known as group membership and leadership election protocols, and the SmartFrog framework contains components that implement such a protocol. Note that the important difference between such protocols and simple discovery protocols is the guarantees of consistency of the information at all servers.

The policy component is only activated on the elected leader and, when given the policy (i.e., constraints referring to the number of daemon replicas), allocates daemons to servers in the group so as to balance load whilst maintaining the constraints defined in the policy.

If a node should fail, this is discovered by the group membership protocol and notified to the policy component, which in turn reallocates the service daemons as required to the surviving servers. If the leader should fail, this will be noticed by the whole group membership and a new leader will be elected. This component will ensure that the service daemons are validly distributed to satisfy the given policy.

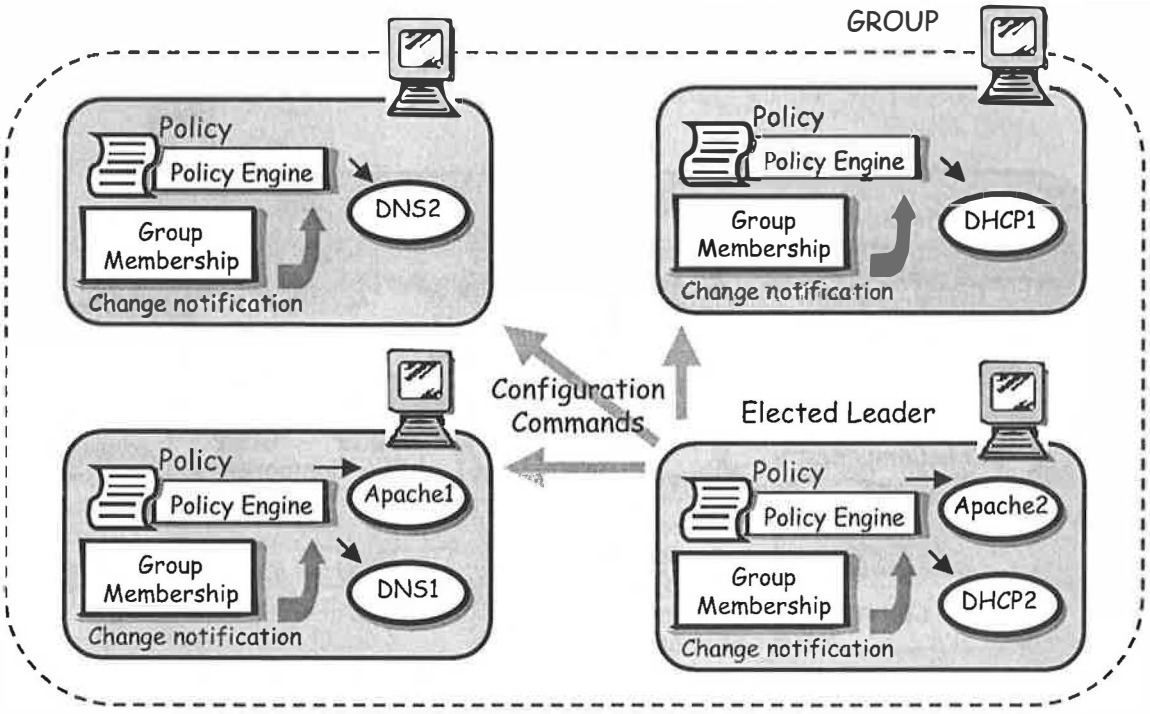


Figure 4: Service reliability.

SmartFrog provides the ability to dynamically describe and manage the configuration requests for the daemons, as well as providing the core components to handle the group communication. LCFG provides the ability to configure the initial collection of servers with the SmartFrog infrastructure, the initial components and, if necessary, the policy description. It also provides the low level components to configure services such as DHCP, DNS, printing, e-mail, and so on that will be triggered by SmartFrog as services are moved around between the servers. Figure 4 illustrates these concepts.

Service Location

The second scenario consists of reconfiguring a system as services move, using a simple service location protocol such as the IETF protocol [18] described in RFC 2165.

A set of Linux file servers offer a set of possibly replicated read-only file systems to a very large collection of Linux client machines via NFS. Each client may require its own unique set of mounts selected from this possible set. Furthermore each file system may be offered by a number of different file servers, with the servers allocated so as to satisfy requirements for reliability, load-balancing of read-requests and storage capacity.

The configuration problem is as follows: although the overall configuration specification for the system may contain the mapping between file-system and server, plus each client's requirements for mounting the various file-systems, changes to the allocation of file-systems to servers may result in many thousands of updates to client machines. These updates would be to modify the automounter settings to mount the correct file servers.

Unfortunately, this is not best handled by pushing these changes to the client machines from some central point as this provides limited scalability and dynamism.[3] A better approach might be to configure a service location component in every client, and a service advertising component into every file server and allowing the distributed service location protocols to resolve the binding between them.

Thus a server would be configured to advertise its own file systems. A client would be told which file systems to locate and prepare an automounter entry for this. Any change of association between server and file-system is then only made on the server and the clients "discover" the new bindings through the location protocols. If more that one server offers access to a specific file system, a number of options exist. A server could advertise itself as the preferred server, in which case the client would select this one in preference. If all servers are equal, a random choice could be made by the client thus spreading load amongst the various servers.

Finally, if a server disappears or communication problems exist between the client and the server (this could be monitored by the client configuration components, for example by scanning log files) a binding to an alternative server could be made. Thus a local decision can be made to resolve a locally identified, localized problem.

Within the combined LCFG/SmartFrog environment, this would be carried out in the following way. LCFG contains the basic components to handle NFS

[3]Using LDAP or NIS for the maps would create a single point of failure in the master database server, and would not solve the problem of updating the maps when a server fails or appears.

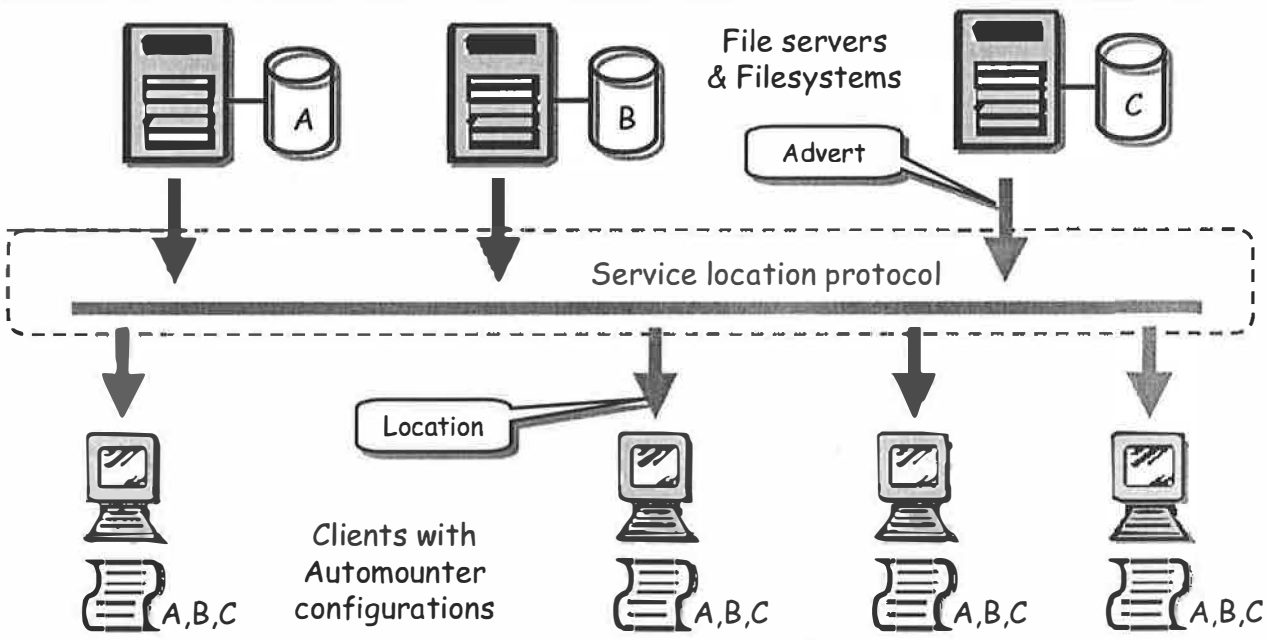


Figure 5: Service location.

servers, configuring the automounter, and so on. SmartFrog provides the appropriate service location and advertising components, using a encapsulated implementation of the SLP protocol. Configuration descriptions would be created that define, for each server and client, the set of file systems they hold, or require. The SmartFrog system would then dynamically instantiate the completed service, using the LCFG components for the specific node configurations. Figure 6 illustrates these concepts.

The GPrint Demonstrator

The GPrint demonstrator is a complete, self-contained cluster that provides a robust printing service via an OGSA interface. This has been developed as part of the GridWeaver project and is described more fully in [10]. It illustrates how the combined LCFG and SmartFrog frameworks can be used to provide a robust printing system that can automatically adjust for print server or printer failures. A short video is available [20] on the web, which demonstrates the GPrint system in action.

The underlying printing system is based on LPRng [3]. The key goals of the system are:

- **Fault tolerance**: no single point of failure combined with autonomic reconfiguration when any part of the system fails.
- **Minimal management**: e.g., to deploy a new print server it should simply be a matter of plugging the node into the network and deploying a print server configuration description to the node.
- Illustrate how the configuration system can integrate with the Globus Toolkit and OGSA standards [15] to provide a robust grid enabled application.

The design of the system is illustrated in Figure 5 and explained further below:

- LCFG provides components to manage the LPRng printing daemon. These components have been used though they are managed by SmartFrog using the SmartFrog → LCFG adaptor.
- A SmartFrog component has been developed to represent a print server. Using LCFG, this component can be deployed on any node. Once deployed, the component advertises the print server using SLP. The rest of the printing system listens for these advertisements and deploys the printing services across the available resources.
- A SmartFrog component has been developed that can scan a range of network addresses for printers. Once a printer is found, a proxy SmartFrog printer component is deployed to represent the printer to the printing framework.

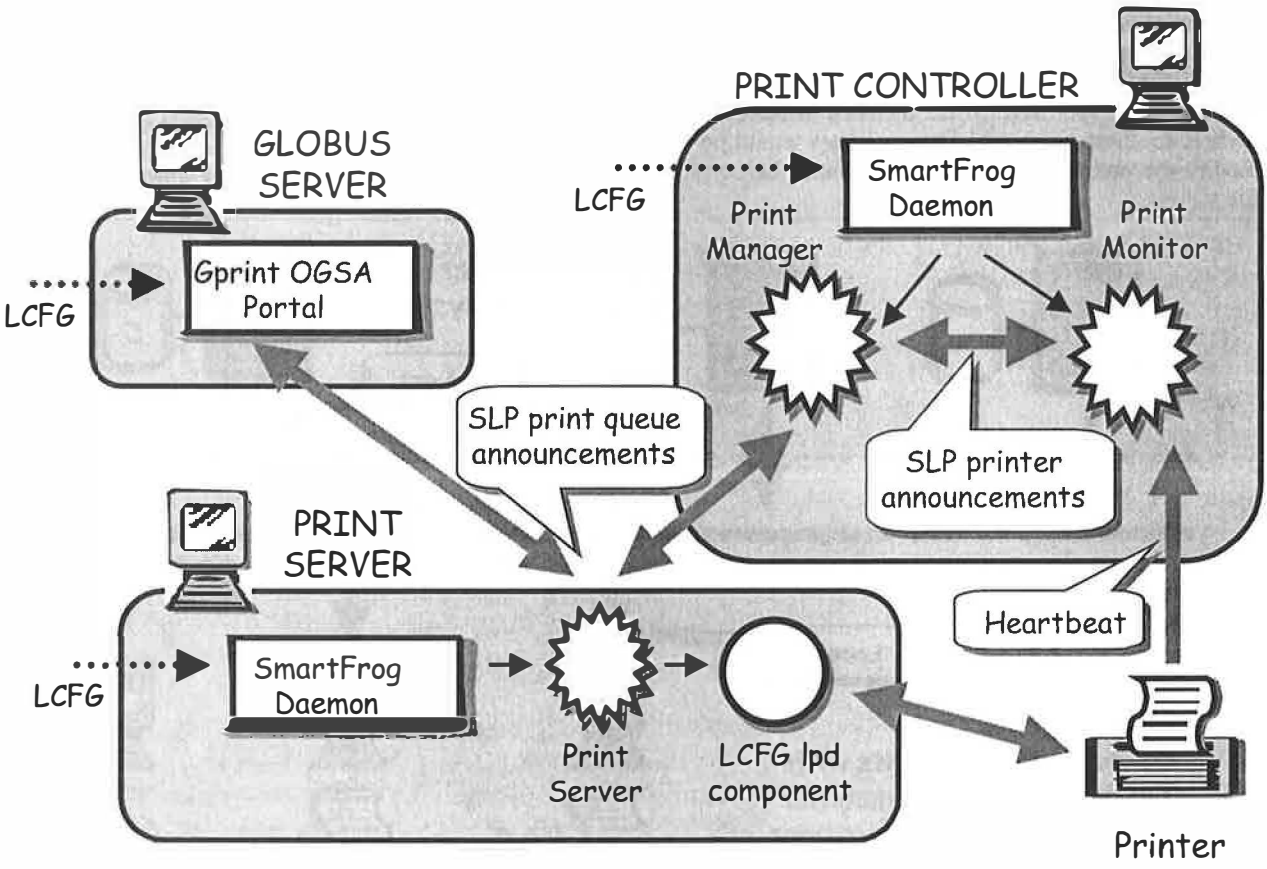


Figure 6: The GPrint demonstrator.

This component performs heartbeat monitoring of the printer and advertises the printers existence using SLP.

- The brains of the system is a SmartFrog print manager component that listens for the print server and printer announcements. This component is configured with a list of the print queues that are to be made available to the end users of the printing system. It deploys these queues based on the currently available resources and will reallocate the queues if any part of the system fails.

  The SmartFrog leadership election protocols can be used to deploy multiple print server managers on different nodes. Only the currently elected leader will perform any system configurations. The other copies are there to take over if the leader node fails. This prevents the manager component from becoming a single point of failure in the system.
- When the print server manager deploys a print queue, this involves deploying a SmartFrog Queue component onto the appropriate print server node. As well as configuring the queue, so that it can print, this component advertises the queue's existence using the SLP protocol. This allows any interested network components to be notified of the current printing configuration. In the full GPrint system, an OGSA portal has been developed which allows print jobs to be submitted through a Grid style interface. The GPrint portal listens for the print queue announcements and advertises the available print queues through its OGSA interface.

Note that the entire GPrint cluster can be rebuilt from "bare metal" machines, using just the configuration specifications and an RPM repository. Likewise, new nodes of any type can easily be installed and incorporated into the cluster.

Conclusions

Future configuration systems will need to incorporate a high degree of autonomy to support the anticipated need to scale and demands for robustness. In practice, this is likely to require several different configuration paradigms, such as explicit specification, or discovery by service location protocol.

We have shown that it is possible to build a configuration framework that allows different paradigms to be incorporated easily, without changes to the components that actually deploy the configuration. The prototype implementation of this framework provides a testbed for experimenting with different configuration paradigms using real, production configurations.

We have also demonstrated that the framework can be used to construct a typical real service with autonomic fault-recovery. The service can easily be restructured to support different modes of configuration.

Acknowledgements

This work has been performed as part of the GridWeaver [13] project, funded under the UK eScience Grid Core Programme. The success of the project is due to the contribution of the whole project team which included Paul Anderson, George Beckett, Carwyn Edwards, Kostas Kavoussanakis, Guillaume Mecheneau, Jim Paterson, and Peter Toft.

Thanks also to Alva Couch for shepherding the paper, and Will Partain for his invaluable advice on the abstract and a regular supply of inspiring ideas.

Availability

LCFG software is available under the GPL from www.lcfg.org, although work is currently underway to improve the packaging, and anyone interested in downloading a current version of the software is invited to contact Paul Anderson at dcspaul@inf.ed.ac.uk.

HP is currently working on a public release of the core SmartFrog framework, including source code. Anyone interesting in obtaining this may contact Peter Toft at peter_toft@hp.com.

The GPrint demonstrator is not intended for production use, but code is freely available by contacting the authors.

Author Information

Paul Anderson (dcspaul@inf.ed.ac.uk) is a principal Computing Officer in the School of Informatics at Edinburgh University. He is currently involved in the development of the School's computing infrastructure as well as leading several research projects in large-scale system configuration. He is the original architect of the LCFG configuration framework.

Patrick Goldsack (patrick.goldsack@hp.com) has been with HP Laboratories in Bristol, England since 1987, where he has worked on a variety of research projects in areas ranging from formal methods, through network monitoring and management techniques, to Grid and utility computing. His research interests are in formal languages and large-scale distributed systems.

Jim Paterson (jpaters2@inf.ed.ac.uk) left the University of Glasgow in 1989 with a Ph.D. in Theoretical Physics. Since then he has worked as a software developer for a number of consultancy firms before taking up his current position as a Research Associate at Edinburgh University. His research has focused on problems in large scale system configuration.

References

[1] *The DataGrid Project*, http://www.datagrid.cnr.it/.

[2] *LCFG*, http://www.lcfg.org/.

[3] *LPRng Printing Framework,* http://www.lprng.org.

[4] Anderson, Paul, "Towards a high-level machine configuration system," *Proceedings of the 8th Large Installations Systems Administration (LISA) Conference*, pp. 19-26, Berkeley, CA, Usenix, http://www.lcfg.org/doc/LISA8_Paper.pdf, 1994.

[5] Anderson, Paul, *The Complete Guide to LCFG*, http://www.lcfg.org/doc/guide.pdf, 2003.

[6] Anderson, Paul, George Beckett, Kostas Kavoussanakis, Guillaume Mecheneau, and Peter Toft, "Technologies for Large-scale Configuration Management," *Technical report, The GridWeaver Project*, http://www.gridweaver.org/WP1/report1.pdf, December, 2002.

[7] Anderson, Paul, George Beckett, Kostas Kavoussanakis, Guillaume Mecheneau, Peter Toft, and Jim Paterson, "Experiences and Challenges of Large-scale System Configuration," *Technical report, The GridWeaver Project*, http://www.gridweaver.org/WP2/report2.pdf, March, 2003.

[8] Anderson, Paul and Alastair Scobie, "Large Scale Linux Configuration with LCFG," *Proceedings of the Atlanta Linux Showcase*, pp. 363-372, Usenix, Berkeley, CA, http://www.lcfg.org/doc/ALS2000.pdf, 2000.

[9] Anderson, Paul and Alastair Scobie, "LCFG – The Next Generation," *UKUUG Winter Conference*, UKUUG, http://www.lcfg.org/doc/ukuug2002.pdf, 2002.

[10] Beckett, George, Guillaume Mecheneau, and Jim Paterson, "The gprint Demonstrator," *Technical report, The GridWeaver Project* http://www.gridweaver. org/WP4/report4_1.pdf, December, 2002.

[11] Burgess, Mark, "Cfengine: A Site Configuration Engine," *USENIX Computing Systems*, Vol. 8, Num. 3, http://www.iu.hioslo.no/~mark/research/cfarticle/cfarticle.html, 1995.

[12] Cons Lionel and Piotr Poznaski, "Pan: A High Level Configuration Language," *Proceedings of the 16th Large Installations Systems Administration (LISA) Conference*, Berkeley, CA, Usenix, http://www.usenix.org/events/lisa02/tech/full_papers/cons/cons.pdf, 2002.

[13] Edinburgh School of Informatics, EPCC and HP Labs, *The GridWeaver Project*, http://www.gridweaver.org.

[14] Jeff Kephart, et al., "Technology challenges of autonomic computing," *Technical Report, IBM Academy of Technology Study*, November, 2002.

[15] The Globus Project, *OGSA*, http://www.globus.org/ogsa/.

[16] Goldsack, Patrick, "SmartFrog: Configuration, Ignition and Management of Distributed Applications," *Technical report, HP Research Labs*, http://www-uk.hpl.hp.com/smartfrog.

[17] IETF ZeroConf Working Group, *ZeroConf*, http://www.zeroconf.org/.

[18] IETF, *Service Location Protocol (svrloc)*, http://www.ietf.org/html.charters/svrloc-charter.html.

[19] Poyner, Todd, "Automating Infrastructure Composition for Internet Services," *Proceedings of the 2001 Large Installations Systems Administration (LISA) Conference*, Usenix, Berkeley, CA, http://www.usenix.org/events/lisa2001/tech/full_papers/poynor/poynor.pdf, 2001.

[20] Toft, Peter, *GridWeaver, The Movie*, http://boombox.ucs.ed.ac.uk/ramgen/informatics/gridweaver.rm and http://boombox.ucs.ed.ac.uk/ramgen/informatics/gridweaver-v8.rm, 2003.

Distributed Tarpitting: Impeding Spam Across Multiple Servers

Tim Hunter, Paul Terry, and Alan Judge – eircom.net

ABSTRACT

This paper describes an Irish ISP's attempts to combat the abuse of resources caused by unsolicited commercial email. We describe the extension of a multicast system, used to implement POP-before-SMTP relaying, to share information about remote mail servers between multiple mail systems. The information may then be used to tarpit abusive servers – placing delays between SMTP protocol answers thus mitigating their impact on our systems. We then examine how effective this has been, and come up with some ideas for future development.

We also discuss building a policy around this and other measures we use to combat spam. An ISP is in the business of sending and receiving mail – this makes slowing or blocking mail a delicate subject.

Introduction & Problem Statement

Unsolicited commercial email (spam) is a problem that, by now, needs no introduction. If you know about email, you know about spam. There are whole books on how to stop it, and it's even on the nightly news [1, 2]. The spam problem from an ISP perspective is also reasonably well known – spam is expensive in terms of time spent receiving it, space spent storing it, and staff months spent dealing with complaints and the technical aspects of cleaning up after it.

Our company, eircom.net, is the ISP division of eircom, the largest telecommunications provider in Ireland. It is also the largest internet provider in Ireland, serving approximately 500,000 customers. To put this in context, a recent survey estimated that around 766,000 adults in the south of Ireland use the Internet at home [3]. As such, any impact on our services is very visible – and is often reported in the media. Our problems with spam are probably fairly average – both the consistent low-level spam that slowly fills up our filesystems, and high-volume assaults[1] that have at least once slowed even our recently retooled mail system solution [4]. However the impact of it, in terms of customer impression and our reputation, is relatively severe.

There are any number of popular solutions for individual and group spam blocking available. The simple ones are a good start – things like: don't run an open relay; don't allow multiple recipients for null sender; and verify that envelope sender contains a valid domain. Yet the spammers seem to have worked around them. Using a blocking list involves handing a fair amount of responsibility and control to a third party – something that would not make for a good response to a customer unhappy with missed mail. Content analysis tools bring up privacy issues [5], cost a lot in processing power, and tend to require tuning for each individual recipient.

Having rejected those options, we looked around for others and came upon tarpitting. The idea, when applied to SMTP, is that when a sender has triggered the tarpit, the SMTP server places an intentional delay between processing of the sender's "RCPT TO <address>" command, and it's "250 OK" response [6]. The tarpit delay is initially triggered by a particular rate of sending mail. The delay can be increased if the sender continues on sending at the maximum rate allowed.

This seemed to us to be a good middle ground – it would dampen the blow of a dictionary attack[2] or other high-volume mailshot from a single source, preventing server overload. If the sender turns out to be a legitimate source, we have not entirely blocked the flow of information to (or from) our customers. In either case, while the server's performance would not reveal anything amiss, any high tarpit delay can be used to trigger an alert to our operations group – enabling them to examine the situation and decide to block the sender entirely, or remove the delay. The general customer base doesn't see a problem, a mistakenly-captured legitimate sender is not entirely blocked, and we remain in control of the situation.

An additional advantage of this approach was that it fitted easily into our existing infrastructure.

Our Environment

We currently run several mail servers in parallel, each providing POP and SMTP services using qmail 1.03 and vpopmail 5.0 for local deliveries [8, 9]. The latter allows us to serve two ISPs (eircom.net and indigo.ie) using the same hardware and software. We

[1] Which seem to frequently come on a Friday evening or over the weekend. Ours is not always a polite adversary.

[2] A dictionary attack is the attempt by a spammer to guess user names on a server by using a large combination of common words, common names, and numbers [7].

use a number of qmail patches from the general community, as well as local modifications to both qmail and vpopmail. An SQL database is used to store provisioning and configuration information.

Our current servers have dual one gigahertz Pentium III CPUs, a gigabyte of memory, and run FreeBSD. Together they handle 1200 messages per minute on average, and many times that at peak times. Traffic is distributed among these systems via a pair of server load balancers. User and domain mail is stored on shared NetApp [10] filers.

One of the services we provide is POP-before-SMTP. This allows a user who is downloading their mail from outside of our IP range to relay mail through our servers for a certain amount of time after a successful POP authentication. As a customer's SMTP connection may reach a different server than the POP connection, potentially only a split second later, this information must be rapidly shared between the servers. So the information about who has POP'd, and when, is passed between servers via multicast,[3] and stored in a shared memory table on each individual server. This table is queried by a small application that runs in the tcpserver[4] exec chain and sets an environment variable if the outside IP is allowed to relay. This is the basis for our distributed tarpit information, and is covered in greater detail in the following sections.

Existing Solutions & Other Work

In October 2002, when we began looking into tarpitting in earnest, we did some searching about for existing solutions – but generally assumed that we'd have to do most of the work ourselves, and that the information to be shared would fit fairly well into the existing POP-before-SMTP infrastructure. We really only came up with one close match – Chris Johnson's tarpit.patch [13]. It's a good starting point, but doesn't implement, or have hooks for, the cross-session and cross-server functionality we need.

In going back to look for prior art for this paper, we came across the 2002 LISA paper on "Spam Blocking with a Dynamically Updated Firewall Ruleset" [14]. While this system wouldn't have worked at our site without changes, it certainly matches well with our desire for a non-permanent/partial blocking mechanism that is easily controlled. The modularity – in passing data in and out of a central agent responsible for determining what is blocked – also matches well with our solution (and the general qmail way of doing things). Elements of their design may well be incorporated into future updates of our system.

[3]Multicast [11] is a network protocol that allows one host to send a packet to a selected set of hosts. It's a middle ground between sending to just one host (unicast), or all hosts on the same LAN (broadcast). An application/host "signs in" to a multicast session by connecting to a designated multicast IP address.

[4]tcpserver is part of D. J. Bernstein's ucspi-tcp package of TCP client-server application tools [12]. We use it as a replacement for inetd, to monitor a port and invoke a program or series of programs for each connection.

POP-before-SMTP Implementation: auth-record, auth-monitord, and auth-lookup

qmail provides POP and SMTP service via two different daemons: qmail-pop3d and qmail-smtpd. In order to implement POP-before-SMTP, we need each instance of the POP daemon to record the IP address and timestamp, of a successful non-local authentication, into a table shared across our mail servers. The timestamp indicates when the entry should expire. When a subsequent connection is made to the SMTP daemon, this daemon must check for the connection IP address in the shared table. If the IP address is found, and the entry has not expired, then the SMTP service will allow relaying.

Like a number of D. J. Bernstein's tools, qmail separates functions into mutually untrusting programs. These often run each other in a chain by having each program do its bit and then exec any remaining command line arguments. Use of root is minimized and programs sometimes run as separate users. This approach keeps programs small, promotes security and modularity and also, incidentally, makes it easy for us to create new programs and add new features without making extensive (or sometimes any) changes to the existing code.

We use this modular approach to split up the POP-before-SMTP tasks: sending out a multicast packet containing the IP address; writing the IP address and a timestamp into a shared memory table; and reading the shared table. The first task happens during the POP session, as a part of the exec chain, which looks like this:

tcpserver → vchkpw → auth-record → qmail-pop3d

- The tcpserver process listens on port 110 and forks a new exec chain for each incoming connection, setting $TCPREMOTEIP, the IP address of the remote system, among other environment variables.
- vchkpw reads the username and password from the network, and checks the entries against our user database. If the username and password are valid, vchkpw does a chdir() into the user's directory and execs auth-record. An invalid username and password, a timeout, or the QUIT command will result in the program exiting and the connection being closed.
- If the IP address is not on our local network, and the user is of a type allowed remote access, auth-record sends a multicast packet containing the IP address in $TCPREMOTEIP, and then execs qmail-pop3d.
- qmail-pop3d takes POP3 protocol transaction state commands (LIST, RETR, DELE, etc.) from the connection and returns the appropriate response.

The second task is performed by a daemon called auth-monitord. This program runs on each mail server

and listens to the multicast session that auth-record talks to. It reads the IP address contained in the multicast packet and writes it, along with a timestamp in the future, into a shared memory table. For example, if IP address 1.2.3.4 establishes an authenticated POP connection on server A at 14:45, auth-record on server A sends a multicast packet that is picked up by servers A, B, and C. Given a 15 minute timeout, the auth-monitord process on these servers will write an entry into their system's shared memory table that looks like this:

```
IP Address   SMTP relay allowed until
1.2.3.4      15:00
```

auth-monitord scans the table on a regular basis to clean out expired entries. The entry above will be removed during the first scan after 15:00. However, if another timestamp containing '1.2.3.4' arrives at 14:55, the timestamp will be updated to 15:10 and the entry will not be removed until after that time.

The use of multicast allows us to have the same information in a shared memory table on each mail server. It also removes any difficulty with race conditions – multiple simultaneous writes are placed into a (not necessarily ordered) queue by UDP/IP. auth-monitord needs no table locking code – it merely pulls entries one at a time from the queue.

Multicast is not a reliable protocol, so there is some chance that the tables will not be exactly the same across the mail servers. In general the chances of this are quite low. However, if a mail server is rebooted, or a new server is started, that server will begin with an empty shared memory table, which could be quite noticeable depending on the timeout period. To rectify this, the newly-started auth-monitord sends a multicast packet asking for a table dump from the other servers. The replies consist of multicast packets containing multiple entries with both IP addresses and their appropriate timeouts.

The final POP-before-SMTP task, looking for an entry in the shared memory table, is performed by auth-lookup. This program is invoked as part of the qmail-smtpd exec chain:

tcpserver → auth-lookup → qmail-smtpd

auth-lookup checks for the IP address, contained in environment variable $TCPREMOTEIP, in the shared memory table. If a table entry is present and has not expired, auth-lookup sets the environment variable $RELAYCLIENT with a null value. If this value is set, qmail-smtpd will allow the incoming connection to relay mail.

Figure 1 continues with our example data, and illustrates the flow of information between mail client and mail servers, and in between the mail servers themselves. In order to simplify the diagram, the only programs shown are auth-record, auth-monitord, and auth-lookup. The POP-before-SMTP process proceeds as follows:

1. At 14:45, the mail client on IP 1.2.3.4 establishes a POP3 connection with mail server A. The client gives a valid username and password to vchkpw, which then invokes auth-record.
2. auth-record sends a multicast packet that indicates that '1.2.3.4' has an authenticated POP3 connection. This packet is received by the auth-monitord process on mail servers A, B, and C.
3. On each server, the auth-monitord program writes an entry into the shared memory table for 1.2.3.4, with a relay expiration time of 15:00.
4. At 14:50, after the mail client has downloaded mail via POP3, it establishes an SMTP connection which is routed to server C.
5. auth-lookup is invoked, which checks for 1.2.3.4 in the shared memory table. As there is an entry, and it has not yet expired, auth-lookup then sets the $RELAYCLIENT environment

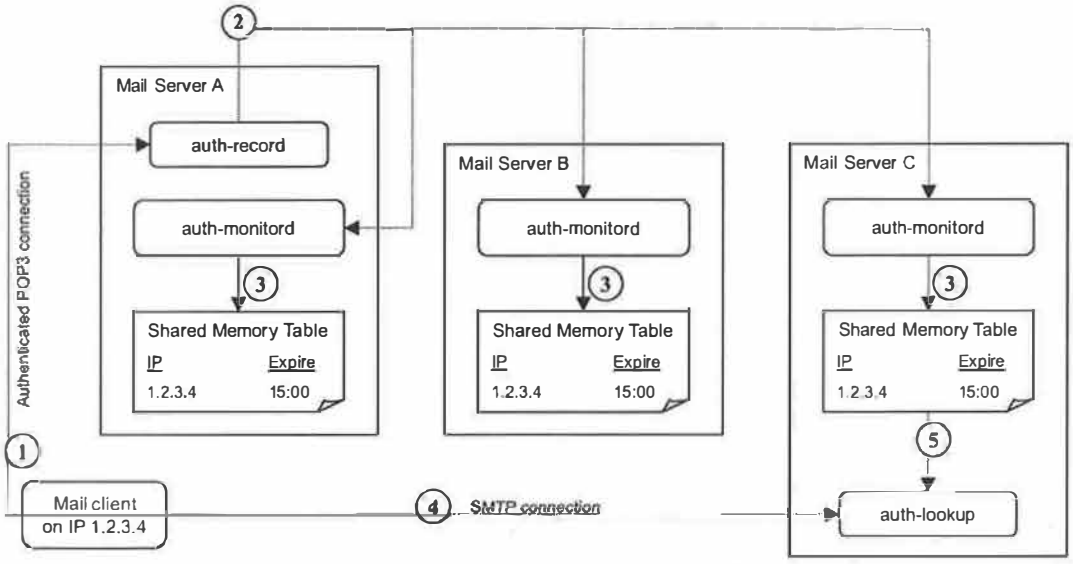


Figure 1: POP-before-SMTP example flow.

variable and invokes qmail-smtpd (not shown). The mail client on 1.2.3.4 will now be able to relay mail through server C.

Tarpit Implementation

Theory

An SMTP tarpit, as we have implemented it, starts off as a simple idea. If a particular remote system has sent us more than *r_max* mail messages, we insert a delay *D* between SMTP RCPT commands from the remote system and responses from our system. Then, as the number of messages continues above *r_max*, we increase delay *D* again once we have received *r_max* + *r_tarpit_inc* messages. We continue to increase *D* each time the total received messages[5] has increased by *r_tarpit_inc*. If we call the received message count *r_count*, we get:

$$\begin{aligned} r\_count < r\_max &\rightarrow D = 0 \\ r\_count = r\_max &\rightarrow D = 1 \\ r\_count > r\_max &\rightarrow D = 1 + Z \end{aligned} \qquad (1)$$

$$\text{where } Z = \text{int}\left(\frac{r\_count - r\_max}{r\_tarpit\_inc}\right)$$

(and the *int*() function returns the number given with any fractional portion stripped away, i.e., int(x) is 1 if ($x \geq 1$ and $x < 2$)).

We also set a maximum value for *D*, so that the SMTP conversation does not violate the relevant standards.[6]

Figures 2 and 3 illustrate how this should work for a single SMTP session. For these examples we've set *r_max* to 1000 messages, and *r_tarpit_inc* to 100. The delay value is zero for the first 1000 messages, and then increases in a linear fashion as more messages are sent in. The impact on how long it takes to send messages to more than a thousand recipients is shown in Figure 3. For this example, we assume that the sender is sending to 4000 recipients, and can send to five recipients per second. As shown, it takes less than four minutes to inject the first 1000 recipients. However, after this the sender is limited to one recipient per second (60 per minute) – and after another 100 recipients, 30 per minute, and so on. As the delay increases, the time needed to insert another 1000 recipients increases exponentially.

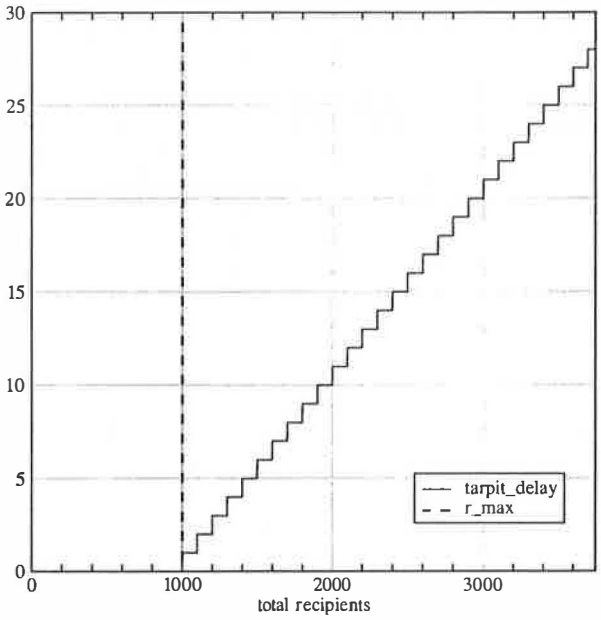


Figure 2: Behavior of Delay D as *r_count* increases.

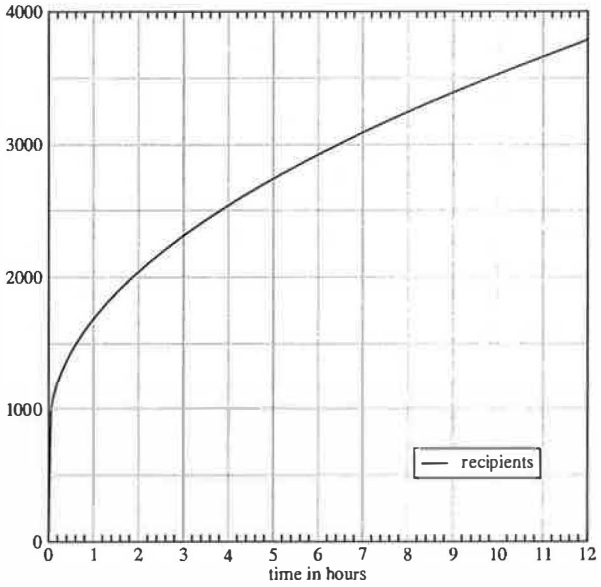


Figure 3: Cumulative recipients accepted over time.

Now, if we extend this idea beyond a single SMTP session, we need to store *r_count* and an IP address across sessions. We'll get into the detail of how this is done below – but you've already got the basics from the discussion of auth-monitord and friends. However, this brings up another problem. Any remote mail server will eventually build up an *r_count* larger than *r_max*, given enough time. We don't want to end up tarpitting every mail server that ever talks to us. It's impolite. Thus we need some way to periodically decrement the *r_count* values – slow enough that we'll catch real spammers, but fast enough that we don't tarpit benign mail servers.

[5]Unless specifically noted otherwise, when we're talking about number of messages in this paper, we're referring to the number of deliveries that the mail system is being asked to make. Someone opening an SMTP connection to our server can insert 100 messages with 1,000 recipients each – and the server will attempt 100,000 deliveries as a result. This is why the delay is inserted at the RCPT command, instead of any other.

[6]Section 4.5.3.2 of RFC 2821 [6] indicates that an SMTP client must wait five minutes for the response to an RCPT command. It goes on to say "A longer timeout is required if processing of mailing lists and aliases is not deferred until after the message was accepted." This could be interpreted to mean that a longer delay would not directly violate the standard.

Here's where things could get quite complicated – an accurate view of how many messages were received during a given time period would require multiple counters and at least one additional time counter. We don't want the shared memory table to be too large, so instead we will store a single count value that we'll periodically modify using two more variables, *r_divide_int* and *r_subtract_int*, in the following equation:

$$r\_count = \text{int}\left(\frac{r\_count}{r\_divide\_int}\right) - r\_subtract\_int \quad (2)$$

Now, we want to be polite and make sure that we don't tarpit every system that ever sends mail to us. However, should a system trigger a tarpit delay, we don't want the reduction operation to let them off too easily. Thus we add another threshold – *r_untarpit*. If a delay is in place, *r_count* must drop below this value in order for *D* to go back to zero. This changes Equation (1) a bit – we now need to talk about D_c (current delay value) and D_n (the next value after we decrement *r_count*).

$$\begin{aligned} &D_c = 0 \;\&\; r\_count < r\_max \rightarrow D_n = 0 \\ &D_c > 0 \;\&\; r\_count > r\_untarpit \;\&\; \\ &\qquad r\_count < r\_max \rightarrow D_n = D_c \\ &\qquad r\_count = r\_max \rightarrow D_n = 1 \\ &\qquad r\_count > r\_max \rightarrow D_n = 1 + Z \\ &\text{where } Z = \text{int}\left(\frac{r\_count - r\_max}{r\_tarpit\_inc}\right) \end{aligned} \quad (3)$$

By this time we've added a bit of complexity to the simple idea of inserting a delay after a certain number of messages are received. However, we should note that the behavior, within an individual SMTP session, is still that of Equation (1), as seen in Figures 2 and 3. At the beginning of the session the values of *r_count* and *D* are the same as the shared values – the session will start with no delay if there is no shared entry, and will start with a delay if both shared values are greater than zero. After this point in the session the delay *D* can only go up – there is no reduction operation, and thus no need for an *r_untarpit* lower threshold. When the session ends, the count of RCPT commands received during that session is added to the shared *r_count* value. Only the shared *r_count* is subject to Equation (2), and only the shared delay D_n is calculated using Equation (3).

We'll see how this looks when we put everything together in the next section.

Data

The theory above has accumulated a few variables that need to be stored in the shared memory tables (the latter having been described in POP-before-SMTP section). Listing 1 is the C structure description of that table.

The *client_address* and *pop_expire_time* variables are simple – the former is the key field and indicates what IP we're describing. If the latter is set and the current time is less than the value, then SMTP relay is allowed from the IP address.

Next we have five variables related to receipt counts. *r_count* and *r_max* describe the current and first threshold recipient count value, as introduced in Equation (1) from the last section. The *r_count* is described as a rough guess based on the inexact nature of our reduction operation. *r_rop_time* specifies the time at which we will next perform the reduction operation detailed in Equation (2). Each time this operation is run, this value is set to the current time plus *r_rop_interval*, defined below. *r_untarpit* is the lower threshold introduced in Equation (3). Finally, *r_last_entry* is a timestamp indicating the last time that the *r_count* value was incremented.

The values *conn_count*, *conn_max*, *conn_rop_time*, *conn_untarpit*, and *conn_last_entry* are used to tarpit based on number of SMTP connections. If *conn_count*

```
struct table_entry {
  struct in_addr client_address;     /* Clients address */
  time_t pop_expire_time;            /* Expire time for POP check */
  int r_count;                       /* RCPT TO's issued (rough guess) */
  time_t r_rop_time;                 /* Time for next reduction op */
  int r_max;                         /* Max value for rcpt count before tarpit */
  int r_untarpit;                    /* Must drop below this to get out of tarpit */
  time_t r_last_entry;               /* For reporting */
  int conn_count;                    /* SMTP connections */
  time_t conn_rop_time;              /* Time for next reduction op */
  int conn_max;                      /* Max value for conn count before tarpit */
  int conn_untarpit;                 /* Must drop below this to get out of tarpit */
  time_t conn_last_entry;            /* For reporting */
  int tarpit_delay;                  /* Determines if and how severely to tarpit */
  int tarpit_max_delay;              /* Max delay for tarpit */
  int config_db_rev;                 /* If the IP is in the config DB.
                                        Values are -1 : error
                                                    0 : ip not in db
                                                  1,2 : alternating db versions */
}
```

Listing 1: Shared memory table entry structure.

goes above *conn_max* during the *conn_rop_interval* (defined below), *tarpit_delay* will be calculated in the same fashion as for excessive recipient counts. If a delay is warranted based on both excessive recipient and connection counts, *tarpit_delay* will be the sum of the two delay calculations.

As you may have guessed, *tarpit_delay* is the shared delay value. It is the starting delay value used in qmail-smtpd, and is recalculated each time that the *r_count* or *conn_count* values are incremented or decremented. However, it will not be set higher than *tarpit_max_delay*, a value customizable per IP address.

The final value stored, per IP, in the shared memory table, is the *config_db_rev*. If greater than zero, the IP address in this table entry is also in the configuration database. The configuration database gives us the ability to set custom values for *r_max*, *r_untarpit*, *conn_max*, *conn_untarpit*, and *tarpit_max_delay*. This allows us to disable tarpitting (or at least set very high thresholds) for systems like our own internal mail servers or other ISP's mail servers.[7] It also allows us to set lower than normal thresholds.

```
CREATE TABLE tarpit_authmonitor_config
(
    r_subtract_int MEDIUMINT NOT NULL,
    r_divide_int MEDIUMINT NOT NULL,
    r_rop_interval MEDIUMINT NOT NULL,
    r_max_default MEDIUMINT NOT NULL,
    r_untarpit_default MEDIUMINT NOT NULL,
    r_tarpit_inc MEDIUMINT NOT NULL,
    conn_subtract_int MEDIUMINT NOT NULL,
    conn_divide_int MEDIUMINT NOT NULL,
    conn_rop_interval MEDIUMINT NOT NULL,
    conn_max_default MEDIUMINT NOT NULL,
    conn_untarpit_default MEDIUMINT NOT NULL,
    conn_tarpit_inc MEDIUMINT NOT NULL,
    pop_timeout MEDIUMINT NOT NULL,
    tarpit_max_delay_default MEDIUMINT NOT NULL,
    notarpit MEDIUMINT NOT NULL,
    shm_table_entries MEDIUMINT NOT NULL,
    last_modified TIMESTAMP NOT NULL,
    created TIMESTAMP NOT NULL
);
```

Listing 2: SQL table definition for global tarpit configuration.

The two legitimate values for this variable are 1 and 2. The differing values are used to update the shared memory tables when a change is made to the configuration database. If all non-zero *config_db_rev* values in shared memory are '1', and the configuration database is updated, its revision will become '2', and the shared memory table will be updated accordingly. The next update will reuse the revision value '1'. While somewhat simplistic, this is sufficient for our needs, as database configuration changes are rare. If the value is less than zero, there was an error while reading the configuration from the database.

[7] We hereafter refer to the practice of setting very high thresholds for a particular IP as whitelisting that IP. Whitelisting can also refer to disabling tarpitting (setting *tarpit_max_delay* to zero) for a particular IP.

Our configuration database contains global settings, in addition to the per-IP custom settings mentioned above. Listing 2 shows the SQL definition for the table that holds these settings. The default values for the customizable settings are stored here (*r_max_default*, *r_untarpit_default*, *conn_max_default*, *conn_untarpit_default*, and *tarpit_max_delay_default*).

The database also stores values for the count reduction operation described in Equation (2) – *r_divide_int* and *r_subtract_int*. How often the reduction operation is performed is determined by *r_rop_interval*. The number of additional recipients needed to increment the tarpit delay, after *r_max*, is *r_tarpit_inc*. These values also have connection count counterparts.

The remaining miscellaneous settings are used to set the POP-before-SMTP timeout (*pop_timeout*), size of shared memory table (*shm_table_entries*), and whether or not tarpitting is in use (*notarpit*). This last value was used initially to give us an idea of what the code would do without actually delaying any SMTP sessions.

Figure 4 shows a series of four graphs that illustrate the theoretical performance of the tarpit code. These graphs were plotted using data from a perl program that simulates input to the mail system and the tarpit response. The following spammer and tarpit values were used:

```
Spam parameters
  Max simultaneous connections      100
  Messages/connection              1000
  Messages/second, per connection     5

Tarpit configuration
  Reduction interval (minutes)       15
  conn_max              (not simulated)
  r_subtract_int                      5
  r_divide_int                        2
  r_max                            1000
  r_untarpit                        100
  r_tarpit_inc                      100
  tarpit_max_delay (seconds)         30
```

Concentrating on the cumulative messages graph first, we see that tarpitting generally works, but perhaps not immediately as we suspect. During the first hour, the overall insertion rate is slightly less than 29/sec (1700/min) – 17 per minute per connection. After the first hour the injection rate settles to just under 3.4/sec. Still not ideal, but enough to keep the load on our servers reasonable. Were tarpitting disabled, the injection rate from this example would be about 250/sec. Over 24 hours the sender manages to inject nearly 400,000 recipients – this would have taken less than an hour without tarpitting.

Before examining some of the other interesting aspects of the system's behavior as a whole, let's look at the other variables graphed, and how they demonstrate various aspects of the system. *r_count* would exactly track the message total graph were it not for the

reduction operation, which we can see happening every 15 minutes. After hour one, when the overall injection rate becomes negligible, we can see the shape of Equation (2) on the *r_count* graph. The *conn_count* graph is quite similar in shape to that of *r_count* – the *tarpit_delay* graph would be as well, except for the limit imposed by *tarpit_max_delay*. We can see the effect of *r_untarpit* in the area where *tarpit_delay* stays at 8 between hours two and three – while the directly calculated *tarpit_delay* would have gone to zero almost immediately, the *r_untarpit* functionality keeps it at the last value calculated before *r_count* went below *r_max*. There's some argument to be made that the *tarpit_delay* should stay at the maximum value achieved until the count goes below *r_untarpit*.

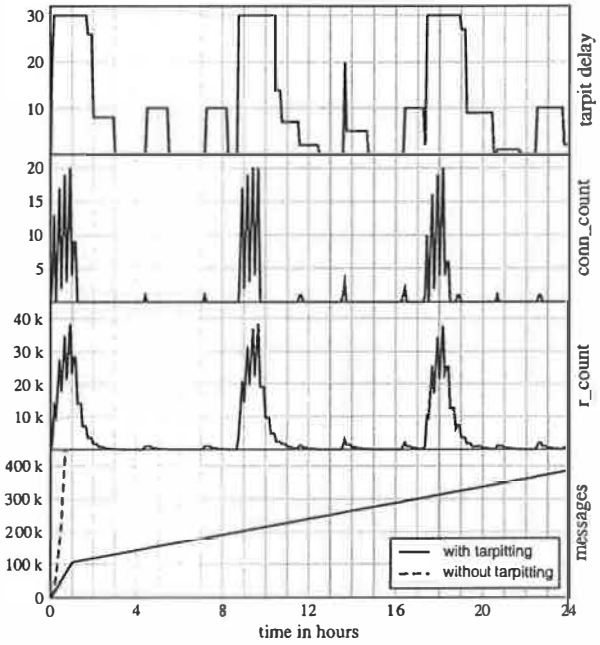


Figure 4: Theoretical example of cumulative messages, *r_count*, *conn_count*, and *tarpit_delay* values over one day.

Finally, why does the injection rate stay low, even when *r_count* and *tarpit_delay* go to zero at hour three? The key to this is the simple tarpit behavior shown in Figure 3. The initial connections that start up in hour zero have no initial tarpit value, and are never tarpitted because they deliver exactly *r_max* messages. However, after ten minutes, any new connections are started up with a *tarpit_delay* value of 30 seconds. While the shared *tarpit_delay* value is reduced every 15 minutes, the individual *tarpit_delay* value in each qmail-smtpd process is not. So each connection started with a *tarpit_delay* of 30 will take just over eight hours to deliver its 1,000 messages!

This is reflected in the *r_count* and *tarpit_delay* graphs – the bulk of the connections time out at eight hour intervals. At these points there are a few sender connections that get a zero *tarpit_delay*, complete quickly, and drive up the shared *tarpit_delay* again for the rest of the restarting connections. There are likely a few connections in each cycle that start when the shared *tarpit_delay* is between zero and 30 – this would account for the intermittent small bursts of activity, spaced two to four hours apart. These connections can be considered to be starting at various points on the Figure 3 graph, and taking as much time as needed to deliver 1,000 messages from that point. All of this serves to stagger out the restarting connections, so that we avoid seeing the 30 messages/sec slope at eight-hour intervals.

We didn't necessarily plan for it to work out this smoothly – the straight lines and periodic behavior are probably more a feature of the simulator than the system itself. Still, somewhat confident that it works well in theory, we proceed to setting it up to work in practice.

Mail System Modifications

As mentioned in the POP-before-SMTP section, our pre-tarpit mail system calls qmail-smtpd as part of an exec chain:

tcpserver → auth-lookup → qmail-smtpd

To implement tarpitting, we need auth-lookup to pass tarpit details to qmail-smtpd, and we also need to be able to write additional details into the shared memory table. So with tarpitting, our exec chain looks like this:

tcpserver → auth-lookup →
qmail-smtpd → auth-record

auth-lookup

auth-lookup checks for an entry in the shared memory table corresponding to the IP address passed by the tcpserver process. It also pulls generic configuration information out of a second shared memory table, copied there from the configuration database by auth-monitord.

The four values that it passes to qmail-smtpd (via the environment) are $TARPIT_RCPT_REMAINING, $TARPIT_DELAY, $TARPIT_MAX_DELAY, and $TARPIT_INCREMENT (*r_tarpit_inc*). The first represents the remaining recipients allowed before *tarpit_delay* is incremented – this is calculated in auth-lookup to minimize the amount of code we're adding to the already weighty qmail-smtpd.

The values passed are based on the values stored in shared memory for the source IP address of the connection, or default values if the IP is not in the main shared memory table. All values are passed as zero if the global configuration variable notarpit is set to 1.

qmail-smtpd

We only make three, rather simple, modifications to qmail-smtpd. The first is pulling the values set by auth-lookup out of the environment. At this point, if *tarpit_r_remaining* and *tarpit_delay* are zero, tarpitting is disabled in this instance of the process. As zero is the default value for these two variables, tarpitting

will also be disabled if the environment variables $TARPIT_RCPT_REMAINING and $TARPIT_DELAY are not set.

The second modification is to the smtp_rcpt subroutine. This is the subroutine that is called to parse the RCPT command. At the end of the subroutine, after all the regular checks and parsing, we add a simple few lines, shown in Listing 3, that actually implement the tarpit.

```
/* tarpitting */
if (r_remaining == 0 &&
       (tarpit_delay == 0 ||
        tarpit_increment > 0)) {
    if (tarpit_delay < tarpit_max_delay)
        tarpit_delay++;
    r_remaining = tarpit_inc;
}
if (notarpit == 0 && tarpit_delay > 0)
    sleep(tarpit_delay);
if (r_remaining > 0)
    r_remaining--;
r_count++;
```

Listing 3: C code in qmail-smtpd's smtp_rcpt subroutine to implement tarpitting.

The first if clause checks to see if the internal *tarpit_delay* count needs to be increased. The increase will happen if *r_remaining* is zero and either *tarpit_delay* is zero (the initial case where *r_count* has reached *r_max*), or *tarpit_increment* is greater than zero. The *tarpit_increment* check is to verify that the configuration is in a sane state. Even if all conditions are set for an increase, *tarpit_delay* is not increased above the *tarpit_max_delay* threshold.

We then simply sleep() for *tarpit_delay* seconds, as long as *tarpit_delay* is non-zero and tarpitting is enabled. Finally, we decrement *r_remaining* and increment *r_count*. Even though one is more or less a reflection of the other, their relationship is complicated enough (see Equations (1) and (2)) to merit recording them separately.

The last modification to qmail-smtpd allows it to send the per-session recipient count information on to the other systems. The main() subroutine is modified to accept a command-line argument (the next program in the exec chain) and store it in the global variable "argl". Then various relevant exit points are modified to write qmail-smtpd's internal *r_count* into the environment variable $TARPIT_RCPT_COUNT, after which the next program in the command chain is invoked with execvp(). As mentioned above, this next program is auth-record.

auth-record

The modifications to auth-record are fairly short. An initial check is made to see if the $TARPIT_RCPT_COUNT environment variable exists and is non-null. If so, a multicast packet is sent containing the recipient count value and a singular increment to the connection count.

auth-monitord

auth-monitord is the daemon that establishes the shared memory table used to store information about POP-before-SMTP. It also records updates, regularly scans the table to expire old entries, and will send out the contents of its table to populate another server's table.

Updated for tarpitting, auth-monitord has three new features: an additional shared memory table to store global configuration settings; global and per-ip settings are read from a database and written to shared memory at regular intervals; and the main shared memory table now stores tarpitting information. Individual *tarpit_delay* values are recalculated after per-IP updates and during each table scan.

Pseudocode for the startup and main loop of auth-monitord is shown in Listing 4. Most of it should be fairly self explanatory – the *tarpit_delay* calculations and reduction operations are as in the Theory section. There are, however, a few details worth noting. The first is that the shared memory table is hashed for faster access, as our current table size is about 16,000 entries (average number of valid entries on a weekday is about 7,600). This uses nearly a megabyte of memory.

auth-monitord is started at boot time, so it's almost certainly the first program on the system to open the multicast session. When opening the socket to be used for multicast communication, we set SO_REUSEPORT on the socket so that the same port may be used by multiple programs. In addition to reads and writes from auth-monitord, one or more copies of auth-record may be writing at the same time.

```
struct auth_message {
  int op_code;
  int table_len;
  int version;
  struct table_entry entry[0];
};
```

Listing 5: Multicast message structure.

Listing 5 shows the one simple message structure used by auth-record and auth-monitord for passing information. The op code is either AUTH_ADD, for passing one table entry (table_entry is defined in Listing 1), AUTH_DUP, for passing multiple table entries, or AUTH_GET, for requesting table entries from other auth-monitords. The routine that sends the contents of a shared memory table does so by sending multiple small packets – each less than the size of the ethernet MTU. The packets are sent with a 1 to 10 millisecond delay between each one, in an attempt to avoid flooding the network. An average update only takes a few seconds.

When an AUTH_GET request is issued, every mail server sends out an update except the one that requested the update. This results in multiple table dumps going over the network at the same time – and all auth-monitords process every update packet. While not terribly efficient, this mechanism is simple to implement, only happens when a mail system is

rebooted, and ensures that all systems are in sync. When a table entry is received, via AUTH_DUP, that already exists in the table, the newer of the *last_entry* times are used, and the entry with the larger (connection/recipient) count prevails, determining both the count and the next reduction op time.

The other source of shared memory table updates, aside from auth-record messages, is the configuration database. The database is re-read about every five minutes, and the *config_db_rev* variable mentioned earlier is used to implement any changes needed to individual table entries. The update routine sets *config_db_rev* to 1 if the previous revision was 2 or an error (zero or less), and to 2 otherwise. It then writes all per-IP configuration settings from the database into the shared memory table, marked with the current *config_db_rev*.[8] When the subsequent database scan is performed, entries with a revision of less than one have values like *r_max* checked against the database default, and updated if the default has changed. If an entry with custom settings is not marked with the current *config_db_rev*, it must have been removed from the database, and is therefore removed from the table.

Finally, it's worth noting that auth-monitord also uses two sanity check routines quite frequently: table_check verifies that shared memory data structure pointers are valid, and is run every time the table is searched or scanned; table_check_entry verifies that a particular entry has expected values, and is run each time an entry is added or updated. If these checks were not performed, a corrupted entry or table could either shut down the SMTP service or turn the system into an open relay – either option is highly undesirable.

[8]This means that the shared memory table always contains an entry for an IP that has custom thresholds in our database. Given that we currently only have 100 custom entries, this isn't yet a problem.

User Interface

A user interface is needed for tarpitting for three reasons – as a debugging tool, to allow changes to the configuration, and to tell us who is being tarpitted so that they may be either whitelisted or blocked from sending mail entirely (blacklisted). Almost coincidentally, we have a tool for each reason: auth-dump, a web interface, and auth-watch.

```
Load initial configuration from database (including shared memory table size);
Establish shared memory table and table data structure;
                /* entries are hashed by last octet in IP address */
Copy configuration settings from database into shared memory;
Build multicast network socket;
Load per-IP configuration information from the database and place it in shared memory;
Send message with op code AUTH_GET to multicast port;
while (TRUE) {
    Sleep until a message arrives or 5 seconds passes;
    if (A message has arrived) {
        Read message from network;
        Discard message if source is a unicast address we don't trust;
        if (message_op_code == AUTH_ADD) { /* single entry from auth-record? */
            Sanity-check entry, add it to the table;
            Recalculate its tarpit_delay ;
        } elseif (message_op_code == AUTH_DUP) { /* multiple entries from another
                                        auth-monitord, in response to an AUTH_GET */
            Sanity check (multiple entries); add them to the table;
            Recalculate tarpit_delay values;
        } elseif (message_op_code == AUTH_GET) {
            Send contents of our shared memory table;
        }
    } /* end if a message has arrived */
    if (It has been five minutes or more since last table scan) {
        Load global and per-IP configuration information from the database;
        Update copy in shared memory;
        /* Scan table: */
        Expire POP times if appropriate;
        Perform connection and receipt-count reduction operations that are due;
        Remove empty entries;
        Insert new default values if appropriate;
        Recalculate tarpit_delay
    }
}
```

Listing 4: Pseudocode for auth-monitord.

- auth-dump does what it says on the tin. For each entry in the shared memory table, a line is printed with the entry values separated by colons. Useful, but not exactly user friendly.
- Our web interface, however, is user friendly. It provides the ability to change global configuration settings, list per-IP entries, as well as add, modify, or delete per-IP entries. It also provides a snapshot view of the current shared memory table, allowing us to see which currently active systems have custom thresholds.
- The final tool, auth-watch, is a perl script that provides a running/historical view of the shared memory table. As seen in Figure 5, it shows the current *r_count* and *tarpit_delay* values and the highest values seen during the run of the program. The 'diff' value shown is the difference between the current *r_count* and the one five minutes ago – indicating how active the sender is.

Tarpitting in Practice

Policy

Much of the problem in detecting and preventing spam is finding the line between what is and is not spam. Similarly, tarpitting must be tuned to find the balance between not catching spammers and unduly impeding legitimate customers. Even if we achieve this goal successfully, what do we do if we find a customer system in the tarpit? Here we must decide between two more extremes: do we immediately set custom parameters, so that the customer is not tarpitted again, or do we assume there's a problem on the customer's system and block their ability to send until they resolve it? If we take the position that the customer is always right – and in turn fail to detect a customer open relay sending mail through our servers, this may result in the ISP's mail servers being added to one of the many distributed block lists. This in turn harms all customers.

Finding these balances is a bit of a thorny problem. It's also clearly not something that technology can solve. The solution, or at least the agreed way to go in looking for it, should involve the people who will have to clean up if things get messy – customer support, sales, and management. To this aim, we developed a policy document before we brought the system live.

A policy document that deals with tarpitting may also want to deal with general questions about the use of SMTP services. Terms like SMTP and blacklisting, and concepts like relaying need to be clearly defined. Some of the questions that such a document might address:

- Who will we relay for?
- Who will we accept mail for?
- Who will be tarpitted?
- What happens when we find someone in the tarpit?
- Adding to and removing from an (internal) whitelist or blacklist.
- Monitoring and reporting.

Having such a policy, agreed upon and signed off, is unlikely to protect you from the thorns – but it should provide something of a cushion.

Performance

The tarpit software has been installed on our production systems since November of 2002. However, it was in measurement-only mode (*notarpit*=1) until mid-February of 2003. At that point the ability to delay mail was enabled, but the thresholds were still set quite high. A few weeks later, we were reasonably sure that the system was working as expected, without any side effects. At this point we lowered the thresholds to realistic values.

Prior to this point we'd begun to see intense bursts of incoming mail on a regular basis. These incidents would drive our delivery attempts up to five or ten times the normal rate, resulting in some combination of: load spikes, mail delays, queues overflowing with undeliverable bounce messages, and/or bursts of mail from our systems to one or more external ISPs. The latter resulted from spam sent to invalid addresses bouncing to a fake return address at the ISP or ISPs in question.

```
Top auth-dump entries [rcpts>50], by rcpt count [log 59870: 1384.7]
Sat Jul 19 02:01:52 2003 - Mon Aug  4 02:32:34 2003

    IP Address                r_count    diff   conn_count   tarpit_delay
 1  17.16.128.142             1675/2943  529       214          7/022
 2  17.18.233.246              365/0782  0         6            0/000
    host.domain.com
 3  17.17.54.155               256/0488  92        256          0/000
    yetanother.net.
 4  192.68.62.241              170/0170  0         3            0/000
 5  17.21.82.39                140/0520  39        5            0/000
 6  192.168.171.156            122/0122  35        122          0/000
 7  17.31.38.230               112/0112  15        3            0/000
 8  192.168.52.10              110/0110  0         1            0/000
    obvious.spammerdomain.name
 9  192.168.48.10              108/0108  12        4            0/000
10  17.17.122.2                100/1168  41        327          0/000
11  17.22.191.99                90/0369  -3        4            0/000
12  17.18.234.10                90/0140  -9        24           0/000
```

Figure 5: auth-watch output.

Since early March we have not been as drastically affected by spam. This does not mean that we have eliminated it entirely. We've even experienced a few more burst incidents, but tarpitting has kept the volume of the bursts down to only 2-3 times our normal traffic.

Figure 6 shows data gathered from an actual spam sender. This set of data conveniently shows what works, and what doesn't work, about tarpitting. In the first two hours of the data set, recipient addresses are sent in at a fairly rapid pace, and tarpitting kicks in after 15 minutes – on average the rate is 7 messages/sec, and more than 40,000 messages are sent in total. The fluctuations in *tarpit_delay* are directly related to the reduction operations on *r_count*.

The sender appears to have noticed that tarpitting is taking place, and then tries to work his way around it. Over three hours, 3000 messages are sent at a rate of about 20/min. Entirely reasonable. Finally, at 6.5 hours, things start to pick up again. Over the next 4.5 hours, mail is sent at a more rapid pace, but tarpitting isn't triggered. The sender doesn't come to our attention except during the last burst at the end. Still, in order to stay under our radar they had to keep their send rate at two messages/sec.

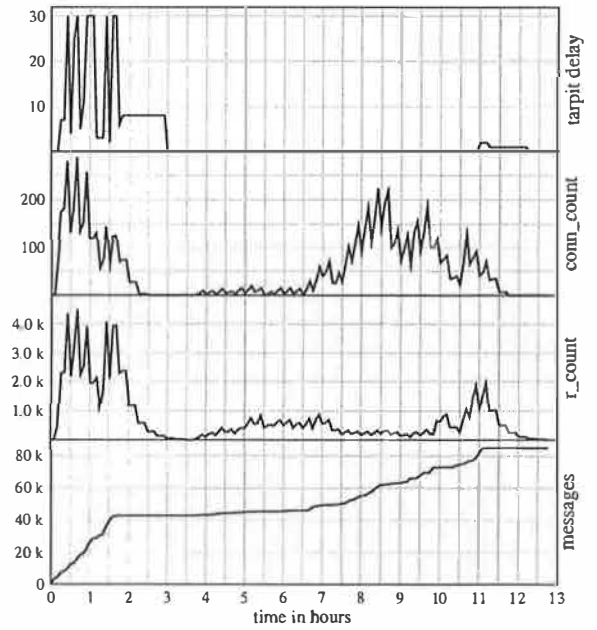


Figure 6: Actual example of cumulative messages and tarpit values for sender seen on July 25.

Overall, the tarpit software is doing a good job of preventing denial-of-service levels of spam. Yet we'd prefer that anybody sending 40,000 messages in a four or five hour period was brought to our attention, and either whitelisted or blacklisted.

Future Work

The first focus area for improvement is the 'grace period' – this is the time period where a spammer opening multiple connections can get a large number of messages in before tarpitting starts. Multiple connections opened at the same time, when the shared *tarpit_delay* is zero, result in (*r_max* × *connections*) messages getting in without being delayed. This also accounts for the initial steep slope in the Figure 4 messages graph, which we have seen in practice. So our first idea for improvement was to ask qmail-smtpd to send incremental updates – instead of updating the shared memory table at the end of a session, update it every time 100 RCPT commands have been issued.

In an "ideal world" situation with lots of connections opening in a short period of time, and messages being sent as quickly as possible for as long as possible, this doesn't make a lot of difference. Under these conditions, data in the shared memory table only benefits new connections – and the bulk of the new connections happen all at the same time as seen in Figure 4. However, if new connections are established in a more random fashion, incremental updates would help ensure that more accurate estimates are available to the connections. In either the ideal or real world cases, incremental updates also give an administrator a more accurate view of the state of the tarpit system.

Another accuracy issue is shared table synchronization across systems – as is, tables are never going to be entirely in sync: each system does its own reduction operations, which won't happen at the same time, and multicast is not a reliable protocol, so there is the potential for lost updates. We've discussed a few ideas for improvement, but generally we've tried to avoid depending too much on strict synchronization – distributed systems are an entire field of study in themselves, and not what we want to concentrate on.

As an alternative solution, we could limit the number of recipients per connection. A related idea would be limiting (across all servers) the number of connections that a remote system is allowed to open at one time.[9] To do this accurately we would need to send a connection-count update at the beginning of the qmail-smtpd session, changing our exec chain:

tcpserver → auth-lookup → auth-record →
qmail-smtpd → auth-record

The first auth-record would be configured to only send a connection count update, and the second to only send a recipient count update, in order to keep the connection counter accurate. The auth-lookup would be configured to exit directly, dropping the connection if the connection count read was at or higher than the maximum. Going back to our theoretical example, we see the following results:

[9]The connection count values in the shared memory table were originally intended to implement connection tarpitting, trapping one-message-per-connection spammers. Observation has shown that our current setup would likely only catch customers with misconfigured mail servers that open multiple SMTP connections without sending anything.

	Messages/sec	
	First hour	Thereafter
Figure 4	30	3.4
Limit recipients per connection by half	16	3.5
Limit connections by half	15	1.8
Both of the above	8.1	2.0

While primarily intended to reduce the initial input rate, these ideas also seem to help with the steady-state rate. Unfortunately our simulator doesn't quite seem to emulate the somewhat less steady traffic as seen in the real world example of Figure 6. One of the ideas that this graph suggests to us, but which would likely show no effect on our simulation, is a less-drastic reduction operation formula. A less-drastic drop (perhaps achieved merely by changing *r_divide_int* to be a floating point value) should remove the rapid fluctuations seen in shared *tarpit_delay* value during the first wave of messages. This also has its downside – the reduction operation is designed to be fairly rapid, in order to avoid penalizing our dynamic IP customers. If one customer were to send mail at a rate high enough to trigger a tarpit delay, and then disconnect, a delay that remained in place for several hours would end up affecting any other customer that was allocated that same IP address. This is clearly something we want to avoid.

We could, however, implement slower reduction operations if we were able to add per-network entries to the database and shared table (as opposed to per-IP entries). We would also make *r_divide_float* one of the customizable values. These changes would enable us to make the default reduction curve longer than that used for our own dynamic-IP networks. This would also enable us to change other defaults for our customers in general – giving us the ability to tighten controls with less risk of reducing service to the customers. A similar idea would be to not only set defaults by network, but tarpit by network – helping us catch spammers who establish multiple connections from multiple contiguous addresses.

A further obvious way to reduce the steady-state input rate, without the need for any code changes, is to increase the *tarpit_max_delay*. Applied to our theoretical example this has no effect on the initial rate, and drops the long-term rate to 1.9 messages/sec. The relevant RFC seems to allow us up to a five minute delay value. These delays not only reduce the incoming rate of spam, but extract a cost – in system resources – from the sender. However, they also extract a cost from us.[10] It might therefore be a good idea to drop all connections from the sender's IP address after they have reached the maximum tarpit delay level for a certain period of time. An extension to this might be a separate timeout (longer than the reduction operation would dictate) for those IP addresses that end up in a drop-all-connections state – making our system a bit more like Deny-Spammers [14].

The final idea we've had on how to improve tarpitting would be to 'seed' the configuration database with information from a network blocking list. Entries could automatically be inserted with thresholds that would cause connections from the listed addresses to be given an immediate delay. Thus mail would still be allowed in, at very slow rates, from senders who may have been listed incorrectly, or who have fixed their spam problem but not yet been removed from the list. It would also give us the local control we desire – an entry in the configuration database with higher-than-normal settings, or a 'do not blacklist' flag, would not be overwritten by a program loading entries from a blocking list.

Conclusion

At the end of 2002 we were faced with an unexpectedly high volume of incoming mail, and with extremely high volume bursts of mail from individual sources. Both problems were largely attributable to a rising tide of unsolicited commercial email. The implementation of tarpitting was intended to dampen the burst attacks, and greatly slow the flow of mail from point sources sending tens or hundreds of thousands of messages. It was also intended to give us a fine degree of control over its behavior, have minimal impact on the customers, and require little change to our existing mail systems.

Tarpitting has definitely helped with the burst attack problem – in the five months since it was turned on, we have not seen any concentrated attacks of the magnitude that we were seeing on a regular basis before that time. Unfortunately, in its current state, it has not solved the general incoming spam problem. As our real world example demonstrated, it's still possible to inject 40,000 messages into our system, in less than six hours, without triggering a tarpit delay. However, we've got many ideas for improving it, and we've even theoretically demonstrated that some will work. Overall it's a useful addition to our spam-fighting toolbox.

There may be other tools out there that would suit us – even tools designed for ISPs much larger than ourselves. One problem with being an ISP is that you are a large target for those sending spam. One can easily see how this would make an organization reluctant to publish any but the vaguest details about how they are fighting spam – much less actual tools. It's clear that those who distribute spam spend as much time finding new ways to send it as we do finding ways to stop it. Thus a pessimistic viewpoint would say that writing this paper just makes their job

[10] One suggestion for fighting the spam problem is to find a method of charging the sender for each message sent. D. J. Bernstein proposes that all mail be stored on the sender's system, or at the sender's ISP [15]. Adam Back's Hashcash system would charge the sender in CPU cycles, by requiring them to produce a token that is difficult to compute but easy to verify [16]. Microsoft's Penny Black research project is investigating several sender-pays techniques [17].

easier. In writing this paper we hope to find a middle ground between hiding everything and full disclosure. Better coordination and information sharing between ISPs (and other large mail volume sites) would lower the volume of spam for everyone.

Availability

How and what code to make publicly available is still under review. If the code is made available, a pointer to it will be available from http://www.qmail.org.

Acknowledgments

The authors would like to thank Erin Hunter and Aoife Cox, as well as Jerry Connolly, Dave Ryan, and Frank Slyne for their assistance in reviewing this paper. We would also like to thank Deeann Mikula for her assistance and shepherding, as well as AEleen Frisch and Rob Kolstad for their incredible patience.

Finally, we would like to thank our management at eircom.net for allowing us to pursue interesting (and productive) ideas, and to share the results.

Author Information

All of the authors have worked at eircom.net for past several years.

Tim Hunter is a systems administrator and manager of a sysadmin team at eircom.net He holds a Bachelor's degree in Electrical and Computer Engineering from the University of Colorado at Boulder, USA. He has worked in systems administration from internal support to on-site customer support in places far away, and currently sits somewhere in the middle. After hours he can occasionally be found running, salsa dancing, hiking, or climbing. He can be reached at tim.hunter@eircom.net.

Paul Terry is a sysadmin at eircom.net. He graduated with a BSc in Mathematics and Mathematical Physics from NUI Maynooth, Kildare, Ireland and has since worked in various roles from operating Macs to coding things for various ISP systems. When not at work, Paul can be found clinging desperately to some cliff shouting at the poor soul who has volunteered to belay him, or running up some mountain, then wondering what the hell he is doing up there and running back down again. Paul can be contacted at paul.terry@eircom.net.

Alan Judge is Head of Research and Development at eircom.net. Most recently concentrating on product development and ISP strategy, he has worked in operations, systems and network administration, campus and ISP architecture, data centre design, and object-oriented distributed systems. He started out as a student admin on the first Unix machine in Ireland and got his first job working for the first ISP in Ireland. He holds Ph.D. and B.A. (Mod) degrees in Computer Science from Trinity College, Dublin, Ireland. He can frequently be found carrying one of the excessive number of cameras he owns or in hiking boots, or both. He can be reached at Alan.Judge@eircom.net.

Disclaimer

References

[1] Schwartz, Alan and Simson Garfinkel, *Stopping Spam*, O'Reilly & Associates, http://www.oreilly.com/catalog/spam/, October, 1998.

[2] CBS Evening News, *No End In Sight For Spammers*, http://www.cbsnews.com/stories/2003/06/18/eveningnews/main559272.shtml, June 18, 2003.

[3] Commission for Communications Regulation, *Quarterly Market Report – Internet Survey*, Ref 03/67c, http://www.comreg.ie/publications/default.asp?ctype=5&nid=101030, May, 2003.

[4] Clark, Matthew, "Spammers Clog Eircom Mail Server," *electricnews.net*, http://www.enn.ie/news.html?code=8678739, November 1, 2002.

[5] Details on Ireland's implementation of the EU data protection act can be found at http://www.dataprivacy.ie. We're honestly not sure that this would apply to direct processing of customer email without the customer's explicit consent – but we wouldn't want to get into an argument about it either.

[6] Klensin, J., Editor, *RFC 2821: Simple Mail Transfer Protocol*, http://www.ietf.org/rfc/rfc2821.txt, April, 2001.

[7] Geller, Tom, "SpamCon Foundation newsletter #013," http://www.spamcon.org/about/news/newsletters/013/opinion.shtml, 11 September, 2001.

[8] Bernstein, D. J., *qmail web page*, http://cr.yp.to/qmail.html.

[9] Inter7 Internet Technologies, *vpopmail web page*, http://www.inter7.com/vpopmail.html.

[10] NetApp, http://www.netapp.com.

[11] de Goyeneche, Juan-Mariano, *Multicast over TCP/IP HOWTO*, v1.0, http://www.tldp.org/HOWTO/Multicast-HOWTO.html, 20 March 1998.

[12] Bernstein, D. J., *ucspi-tcp web page*, http://cr.yp.to/ucspi-tcp.html.

[13] Johnson, Chris, *tarpit.patch*, http://www.palomine.net/qmail/tarpit.patch.

[14] Mikula, Deeann M. M., Chris Tracy, and Mike Holling, "Spam Blocking with a Dynamically Updated Firewall Ruleset," *LISA 2002.*

[15] Bernstein, D. J., *Internet Mail 2000 web page*, http://cr.yp.to/im2000.html .

[16] Back, Adam, *Hashcash – A Denial of Service Counter-Measure*, http://www.hashcash.org/hashcash , 1 August, 2002.

[17] Microsoft Research, *The Penny Black Project web site*, http://www.research.microsoft.com/research/sv/PennyBlack/ .

Using Service Grammar to Diagnose BGP Configuration Errors

Xiaohu Qie – Princeton University
Sanjai Narain – Telcordia Technologies

ABSTRACT

Often network components work correctly, yet end-to-end services don't. This can happen when configuration parameters of components are set to incorrect values. Configuration is a fundamental operation for logically integrating components to set up end-to-end services.

Configuration errors frequently arise because transforming end-to-end service requirements into component configurations is inherently difficult. Such transformations are largely performed in a manual and localized fashion, resulting in high cost of network operations.

The *Service Grammar* technique has been developed to solve the configuration error diagnosis problem and, more generally, to formalize the process of building complex systems via configuration.

At its core is a *Requirements Language* that contains global, high-level constraints upon configuration parameters. These are derived from identifying the notion of "correct configuration" associated with different protocols. These notions are composed to create system-wide requirements on architecture and policies. A *Diagnosis Engine* checks if constraints in the Requirements Language are true given definite component configurations and recursively checks composite requirements.

This paper describes an application of Service Grammar to diagnosing BGP configuration errors. As BGP architecture and policies differ widely from one network to another, it is not possible using previous techniques to check if router configurations implement the *intended* requirements. Our tools enable administrators to specify system-wide, network-specific requirements and check if they are correctly implemented by component configurations.

Introduction

Traditional network management systems diagnose hard, localized errors such as fiber cuts or hardware/software component failures. It is quite possible, however, that network components work correctly yet end-to-end services don't. This happens if there are configuration errors, i.e., configuration parameters of components are set to incorrect values. Configuration is a fundamental operation for integrating components to implement end-to-end services. Configuration errors arise frequently because transforming end-to-end service requirements into configurations is inherently difficult: in realistic networks there are many components, configuration parameters, values, protocols and requirements. Yet, such transformation is largely performed manually. The resulting high cost of network operations as well as the potential for security breaches is well documented [1, 2].

The *Service Grammar* [3, 4, 5, 6] technique has been developed to solve the configuration error diagnosis problem, and more generally, to formalize the process of building complex systems via configuration. At its core is a *Requirements Language* that contains *global*, high-level abstractions that are set up in the process of setting up end-to-end services. A good heuristic for deriving this language is to ask the question "what does it mean for a group of agents executing a protocol to be correctly configured." This language is created for all of the protocols in a domain of interest. End-to-end service requirements can be naturally defined as logical conjunctions of requirements in the language at and across different protocol layers.

This is done by rules of the form A:-B_1, ..., B_k, $k \geq 0$, where each A and B_i is a requirement. A *Diagnosis Engine* checks if a language requirement is true given definite system configuration. By recursive use of this operation, complex algorithms for diagnosis can be developed. Service Grammar captures the intuition to regard a system not as a set of components but as a set of services that, in general, span multiple components.

The information flow of the diagnosis system is illustrated in Figure 1. The diagnosis engine takes input from two sources: (1) service requirements expressed in the requirements language, and (2) vendor-neutral component configurations stored at a centralized database, e.g., an LDAP directory. Raw component configuration is parsed into vendor-independent data structures by vendor-specific adaptors. The diagnosis engine queries the component configuration database and verifies if the configurations are consistent with service requirements. If not, it notifies the administrator where the diagnosis process had

failed. The administrator can then modify the configuration settings and rerun the diagnosis process.

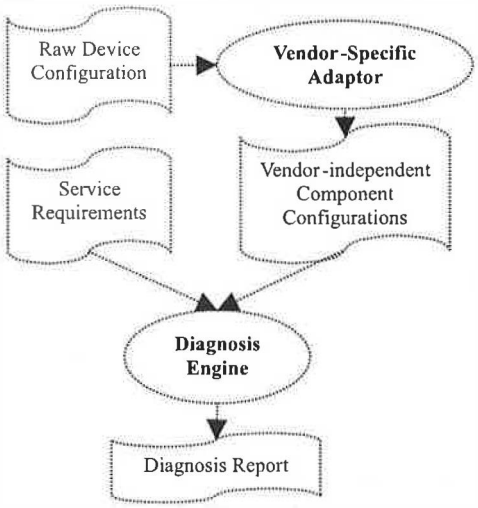


Figure 1: Diagnosis system.

Service Grammars have been built and used for adaptive Virtual Private Networks and mobile security [3, 4, 5, 6]. This paper describes an application of Service Grammar to diagnosing configuration errors in BGP [7]. Previous solutions to diagnosing configuration errors have been network invariant [8] in that they contain a fixed set of constraints that must be satisfied by every BGP network. However, BGP requirements such as logical architecture and policies differ widely from one network to another. It is not possible in previous techniques to check if component configurations implement the *intended* requirements. Our tools enable administrators to specify network-specific requirements and check if they are correctly implemented by component configurations.

BGP Background

BGP is the Internet's inter-domain routing protocol run between autonomous systems (ASes). Routers in different ASes use BGP to exchange information on how to reach destinations throughout the Internet. BGP is path-vector based. A BGP route consists of a network prefix N and an *AS_PATH* of the form $\{AS_k, \ldots, AS_0\}$, which is the ordered list of ASes to traverse to reach N. The AS path is constructed by successively propagating reachability information: each AS prepending its own AS number to the path (one or more times) before sending it to neighbors. Figure 2 illustrates how routing information about network *200.12.0.0/16* is propagated between ASes. For instance, AS160 knows its traffic will traverse AS172, AS180 and AS200 before reaching the destination.

BGP is capable of enforcing policies based on various preferences and constraints. BGP policies affect the route selection and export process, thereby controlling how traffic enters and leaves an AS. Each AS can define BGP policies according to its own criteria. BGP chooses the best route based on a number of metrics, such as the AS path length. In Figure 2 AS172 chooses {180, 200} over {190, 200, 200} as the best route to N because its policy favors a shorter AS path.

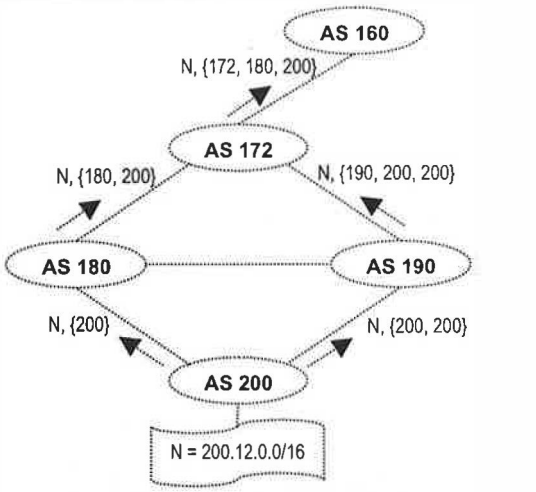


Figure 2: BGP network example.

Policy can be also applied to the route propagation process. An AS decides what to tell its neighbors. If an AS is unwilling to carry certain traffic for a neighbor, its policy will disallow routing advertisements about particular destinations being sent to the neighbor. For instance, AS180 and AS190 chose not to export routes to N to each other. As a result, the horizontal link between the two ASes will not be used to carry traffic to N. In a less restrictive case, AS200 tells AS190 about N, but prepends its AS number twice to make the path longer, indicating the route is considered a less attractive one. The policy eventually affects AS172's route selection process: it chose AS180 instead of AS190 as the next hop AS to reach N.

Challenges of Setting Up BGP

To set up BGP, network administrators configure individual routers in the AS using a configuration language. The following is a sample configuration in Cisco CLI format [9] for a router in AS160. The configuration involves originating routes, establishing peer relationship with neighbors, and applying policy filters. In this example, the router announces network *172.1.1.0/24* and peers with a remote BGP router in AS172. The policy filter allows only routes with an empty AS path (i.e., locally originated routers) to be advertised to AS172:

```
router bgp 160
    172.1.1.0 mask 2525.255.255.0
    network 172.16.24.1 remote-as 172
    172.16.24.1 filter-list 1 out
!
ip as-path access-list 1 permit ^$
!
```

As BGP is a complex protocol, manually configuring individual routers is a time-consuming and error prone task. This is especially challenging in a large network with hundreds to thousands of routers. To maintain a consistent view of routing inside an AS, all BGP routers must be correctly configured to form a full-mesh or some well-structured internal hierarchy, such as route reflector clusters. At a lower-level, two BGP speakers must be able to talk to each other in order to exchange routing information. This seemingly obvious requirement has certain intricacies due to the fact that BGP relies on pre-existing connectivity provided by Interior Gateway Protocols (IGP) or static routes. For example, the remote peer address specified by a BGP router must match the outgoing interface of the IGP route used by the remote peer. Otherwise the connection will not be established, unless the remote peer explicitly specifies the matching interface. This type of implicit requirement can be easily overlooked by administrators, or violated due to change of the network.

Policy routing is an important functionality of BGP, but also provides numerous opportunities for configuration errors. In face of this type of errors, BGP may continue to operate, but does not enforce the intended policy. Policy violation could lead to connectivity, security and economic problems. A well-know problem is address space hijacking, in which one AS accidentally announces networks "owned" by other ASes, forming a "blackhole" within the Internet. Other policy problems are commonly related to the commercial relationships an AS participates in. A multi-homed AS, for instance, shouldn't provide transit service to non-local traffic. The causes of errors are diverse, ranging from typos to poor

BGP Requirements Language	
Requirement	**Meaning**
ibgp_session (RouterA, RouterB, LocalAS)	Both RouterA and RouterB are BGP speakers of the local AS. A BGP session can be successfully established between them.
ebgp_session (LocalRouter, LocalAS, RemoteRouter, RemoteAS)	A BGP session can be successfully established between the local BGP speaker and the remote BGP speaker.
reflector_client_session (ReflectorRouter, ClientRouter, LocalAS)	In addition to ibgp_session requirements, the reflector is set up to forward routing updates from other IBGP peers to the client.
cluster (Reflectors, Clients, LocalAS)	Reflectors and clients form a cluster, i.e., reflectors are fully meshed and all clients are peered with all reflectors.
as_full_mesh (Clusters, Non-clients, LocalAS)	Clusters and non-clients form an AS, i.e., all reflectors and non-clients are fully-meshed. No client peers with a non-client.
route_originate (Subnets, LocalAS)	The LocalAS originates routes represented by subnets.
link_to_provider (LocalRouter, RemoteRouter)	The session represents a link to the local AS's provider. On this session, the local AS should accept everything, but only announce its own routes.
link_to_customer (LocalRouter, RemoteRouter)	The session represents a link to the local AS's customer. On this session, the local AS should announce everything, but only accept the customer's routes.
link_to_peer (LocalRouter, RemoteRouter)	The session represents a link to the local AS's peer. On this session, the local AS should only announce its customers' routes, and only accept the peer's customers' routes.
provider_as (LocalAS, RemoteAS)	RemoteAS is a provider of LocalAS.
customer_as (LocalAS, RemoteAS)	RemoteAS is a customer of LocalAS.
peer_as (LocalAS, RemoteAS)	RemoteAS is a peer of LocalAS.
preferred_outgoing_link (LocalRouter, RemoteRouter, RemoteDestination)	The session is the preferred outgoing link to reach a remote destination, expressed in either subsets or ASPath.
preferred_incoming_link (LocalRouter, RemoteRouter, LocalDestination)	The session is the preferred incoming link to reach a local destination, expressed in either subsets or ASPath.
preferred_neighbor_entry (LocalRouter, RemoteRouter, LocalDestination)	The session is the preferred entry from the neighbor AS to reach a local destination, expressed in either subsets or ASPath.

Table 1: Service grammar for BGP.

understanding of configuration semantics. An excellent empirical study of BGP policy configuration errors is presented in Reference [10].

Service Grammar for BGP

Correct BGP configuration means all routers in an AS achieve the joint goal of exchange routing information, maintaining a consistent view of routing and enforcing intended policies. However, there is quite a large conceptual gap between this global requirement and individual router configurations. Configuration errors arise because manual compilation of these high-level requirements into low-level "machine language" is difficult. If for some reason the global requirements are not satisfied there are no systematic tools to automatically diagnose configuration errors. Network administrators today manually perform these tasks.

Diagnosing why routers don't work together requires global reasoning about the logical structure of the network as well as dependencies between services. The BGP Service Grammar captures these global abstractions that are set up in the process of constructing routing services. By making these definitions explicit, network administrators can formally state high-level network-specific requirements and policies using these definitions.

A subset of the requirement language is shown in Table 1 followed by detailed explanations. The requirements fall into two categories: connectivity and policy.

Connectivity Requirements

The language provides two basic primitives – ibgp_session and ebgp_session – for describing BGP neighbor relationships. We outline the diagnosis procedure for ibgp_session in Figure 3.

Regarding establishing BGP neighbor relationship, the types of configuration errors that can arise include:

- Incorrect AS number or neighbor address at two session end points, peer values are not mirror images of each other (usually typos).
- The neighbor's address is not reachable via IGP. This can happen when a loop-back interface is used but that interface does not participate in any IGP.
- A router tries to connect to a reachable interface of a remote neighbor, but the neighbor uses a different outgoing interface in the reverse IGP route. This happens when the neighbor has multiple reachable interfaces.

Any of the above errors can lead to connectivity problems preventing the BGP session from being established. This example demonstrates that even a very basic BGP requirement implies a number of assumptions and global relationships that the administrator must keep in mind and configure correctly on every router. There are many places for errors. The diagnosis engine systematically validates these assumptions and global relationships, catching all potential errors and providing useful information for the debugging process.

ibgp_session(RouterA, RouterB, LocalAS)

- Meaning: Both RouterA and RouterB are BGP speakers of the local AS. A BGP session can be successfully established between them so they can exchange routing information.
- Diagnosis procedure:
  1. RouterA.as_num == LocalAS
     RouterB.as_num == LocalAS
  2. $\exists$ Ia, Ib, Na, Nb, s.t
     - Ia $\in$ RouterA.interfaces
     - Ib $\in$ RouterB.interfaces
     - Na $\in$ RouterA.neighbors
     - Nb $\in$ RouterB.neighbors
     - Ia.ip_addr == Nb.peer_address
     - Ib.ip_addr == Na.peer_address
     - Na.remote_as == LocalAS == Nb.remote_as
     - RouterA has an IGP route Ir_a to reach Ib
     - RouterB has an IGP route Ir_b to reach Ia
     - If Ia is a loopback interface then Na.update_source == Ia, else Ia is the outgoing interface of Ir_a
     - If Ib is a loopback interface then Nb.update_source == Ib, else Ib is the outgoing interface of Ir_b

Figure 3: IBGP session diagnosis procedure.

cluster(Reflectors, Clients, LocalAS)

- Meaning: Reflectors and clients form a cluster, i.e., reflectors are fully meshed and all clients are peered with all reflectors.
- Diagnosis procedure:
  1. $\forall$ A $\in$ Reflectors, $\forall$ B $\in$ Clients, reflector_client_session(A, B, LocalAS) is TRUE
  2. $\forall$ X, $\forall$ Y (X $\neq$ Y) $\in$ Reflectors ibgp_session(X, Y, LocalAS) is TRUE
  3. All reflectors have the same cluster_id
  4. $\forall$ C $\in$ Clients
     $\forall$ N $\in$ C.neighbors
     If (N.remote_as == LocalAS) &&
     (N $\notin$ Reflectors $\cup$ Clients)
     return FALSE;
     /* Clients shouldn't have IBGP sessions to non-clients */

Figure 4: Cluster diagnosis procedure.

Notice these basic primitives are already higher-level than raw router configurations. They can be used to compose other higher-level requirements that specify an AS' logical structure, such as reflector_client_session and cluster.

Figure 4 illustrates how to validate if a group of routers form a cluster. The algorithm verifies three global properties: all reflectors are fully meshed, all clients can receive updates from all reflectors, and every client only has BGP sessions with routers in the

same cluster. IBGP session test is embedded as part of the procedure.

Policy Requirements

Routing policies are configured via policy filters. A filter consists of a match criteria and a set of actions. Nearly all attributes of a routing update can be used to specify the match criteria, with AS path and network prefix being the most common ones. When a routing update satisfies the conditions set in the match criteria, associated actions (permit, deny, or modify) are invoked to control the propagation of the update. A filter can be applied to route origination, import and export process. It serves as the low-level building block for composing arbitrary routing policies. Our BGP Service Grammar supports the direct use of low-level filters to specify policy requirements. What we highlight in this section is the grammar that describes global AS-level properties, rather than that of an individual filter. These abstractions give network administrators a set of templates for defining common AS routing policies at high-level. Low-level filters can then be used for further refinement. We believe such a design would largely reduce the need for administrators to go into the low-level configuration details of each filter.

link_to_provider(LocalRouter, RemoteRouter)

- Meaning: The session between LocalRouter L and RemoteRouter R is a link to the provider of LocalAS.
- Diagnosis procedure:

```
Let P = Get_provider_AS(L.as_num) ∪
        Get_peer_AS(L.as_num)
Let N ∈ L.neighbors corresponding to R
∀ p ∈ P
  Construct a route update r,
    let r.as_path="_p_"
  if (Routemap_Eval(N.map_out, r)
      ≠ DENY)
    return FALSE;
  /* LocalAS shouldn't leak routes learned
       from providers back to providers */
```

Figure 5: Link to provider diagnosis procedure.:

Typical commercial relationships between two neighboring ASes can be characterized as *customer*, *provider*, or *peer*, as defined in [11]. To test if a remote AS is a customer (provider, or peer) of the local AS, we need to verify that the relationship holds on all sessions between the two ASes. Figure 5 outlines the diagnosis procedure for link_to_provider. When exporting routers to a provider, an AS exports its own and its customer routers, but usually does not export routes learned from providers or peers. A properly configured export filter on this session should block those routes. For each provider and peer AS, the diagnosis procedure constructs an AS path containing the AS number, and feeds it to the export filter. Any of these paths passing the filter is a violation of the policy. In that case, the diagnosis procedure fails. This procedure uses several utilities functions. Get_Provider_AS returns the set of ASes that are marked as a provider of the local AS. Routemap_Eval mimic the processing of a policy filter on a route update.

When multiple routes to a remote destination (network or AS) exist, one link is usually designated as the primary route and others serve as backup. Such a policy can be expressed with preferred_outgoing_link. Its diagnosis procedure (Figure 6) examines the import filter on all EBGP sessions. For each session, the procedure calculates the **local-preference** that a route update to the remote destination would get if it arrives on this session. To pass the test, the import filter on the preferred session must be the one that generates the highest **local-preference**.

preferred_outgoing_link(LocalRouter, RemoteRouter, RemoteDestination)

- Meaning: The session between LocalRouter L and RemoteRouter R is the preferred link for outgoing traffic to RemoteDestination D.
- Diagnosis procedure:

```
Let S = Get_EBGP_Sessions(L.as_num)
Construct a route update r, let r.NLRI = D
Let N ∈ L.neighbors corresponding to R
Let H = Routemap_Eval(N.map_in, r).local_pref
∀ s ∈ S, s ∉ {L, R}
 if (Routemap_Eval(s.local.map_in, r).local_pref > H)
  return FALSE;
  /* Another session is more preferable to this one */
```

Figure 6: Preferred outgoing link diagnosis procedure.

Both preferred_incoming_link and preferred_neighbor_entry are used to control incoming traffic by designating a primary route to a local destination. The difference is that the latter only concerns two neighboring ASes. The diagnosis procedures are similar. Both procedures examine export filters, except that the former looks for the filter that generates the shortest AS path, while the latter looks for the one that generates the lowest multi-exit-discriminator (**med**).

Sample Network Study

We have designed an experimental BGP network consisting of nine CISCO routers in five ASes, shown in Figure 7. The goal is to demonstrate different BGP architecture, peering relationship and routing policies.

Under this setup:

- AS172 represents a large service provider with four routers, three of which are BGP speakers. The three BGP speakers form a cluster with **PR3** being the reflector. Therefore IBGP peering between **CR3** and **CR4** is not required. Inside the AS OSPF is running as IGP.
- AS160 represents a small customer ISP. It connects to the Internet solely via AS172, and thus it is a stub.
- AS180 and AS190 represent two intermediate level service providers. They subscribe service from AS172 and provide connectivity for AS 200. They also enter a bilateral peering agreement.

- AS200 represents a multi-homed customer ISP with 2 BGP speakers. It has multiple links to AS180 and AS190.

Suppose AS200's network administrator wants to enforce the following policies:

1. AS200 announces two networks: *200.12.1.0/24* and *200.12.2.0/24*
2. AS200 is a multi-homed AS. AS180 and AS190 are its providers.
3. AS190 is the preferred AS for outgoing traffic to AS172.
4. **BGP1** is the preferred Border Router for outgoing traffic to AS180.
5. **BGP1** is the preferred ingress Border Router for traffic to network *200.12.1.0/24* from AS180.
6. **BGP2** is the preferred ingress Border Router for traffic to network *200.12.2.0/24* from AS180.
7. AS180 is the preferred AS for all incoming traffic.

To realize these policies, network administrators first need to analyze the underlying requirements that support them. Typically, Policy 1 requires the two networks be originated by the two BGP speakers. Policy 2 requires outbound filters on every EBGP session that only allows locally originated routes to be advertised. Policy 3 and 4 require inbound filters to set up the **local-preference** attribute correctly. More precisely, routes to AS172 learned from AS190 should be given a higher **local-preference**, as should routes to AS180 learned via **BGP1**.

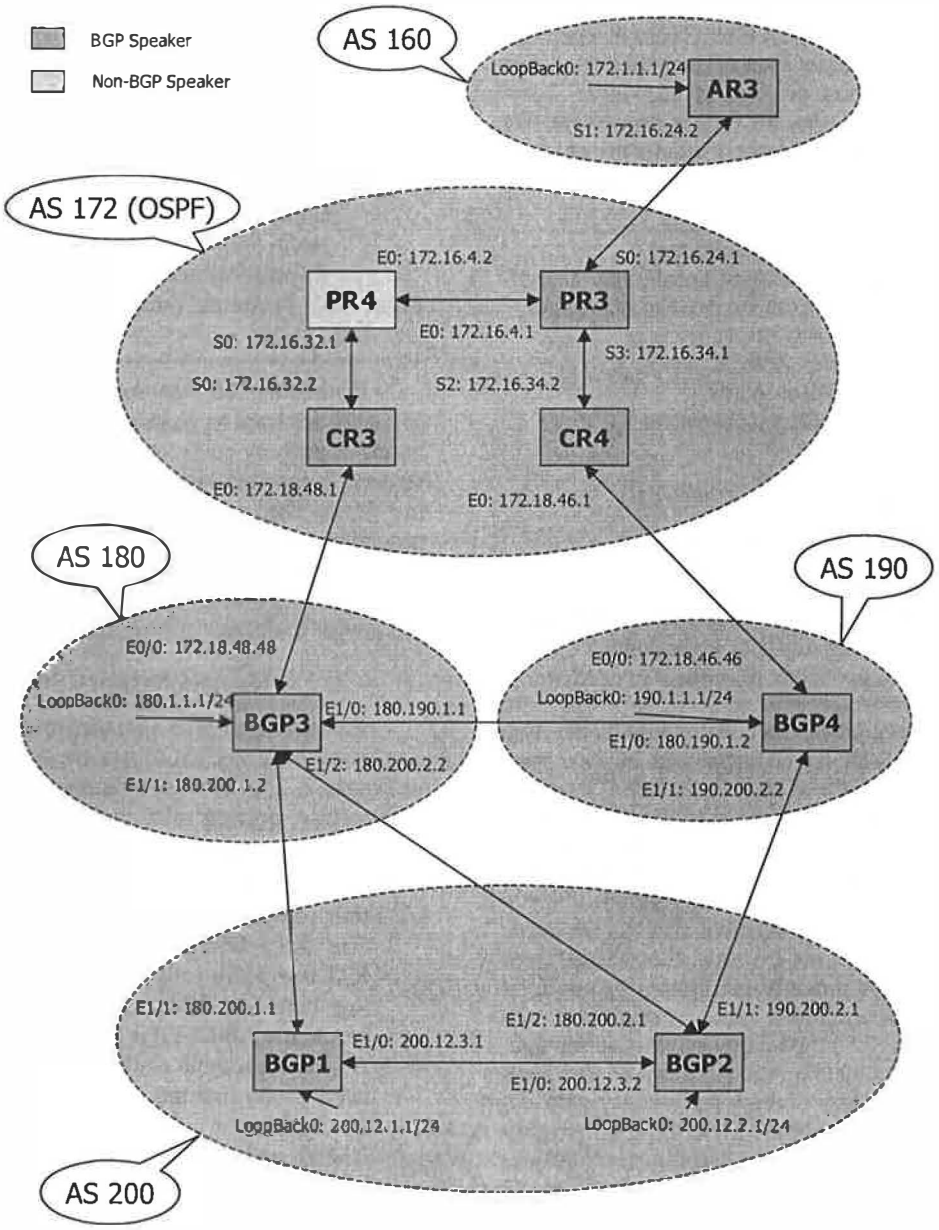


Figure 7: Experimental setup.

```
!
hostname BGP1
!
router bgp 200
   no synchronization
   network 200.12.1.0
   neighbor 180.200.1.2 remote-as 180
   neighbor 180.200.1.2 route-map SETLOCALIN in
   neighbor 180.200.1.2 route-map SETMEDOUT out
   neighbor 180.200.1.2 filter-list 1 out
   neighbor 200.12.2.1 remote-as 200
   neighbor 200.12.2.1 update-source Loopback0
!
ip as-path access-list 1 permit ^$
ip as-path access-list 2 permit 180$
!
access-list 1 permit 200.12.1.0 0.0.0.255
route-map SETLOCALIN permit 10
   match as-path 2
   set local-preference 400
!
route-map SETLOCALIN permit 20
   set local-preference 100
!
route-map SETMEDOUT permit 10
   match ip address 1
   set metric 10
!
route-map SETMEDOUT permit 20
   set metric 20
!
ip route 200.12.2.0 255.255.255.0 200.12.3.2
!
```

Figure 8: **BGP1** configuration.

Policy 5 and 6 requires manipulation of the **med** path attribute. Both **BGP1** and **BGP2** advertise network *200.12.1.0/24* and *200.12.2.0/24* to **BGP3**. **BGP1** should give a more favorable **med** value to network *200.12.1.0/24* than to *200.12.2.0/24*, and **BGP2** should do the opposite. **BGP3** then is able to decide which router is the best to reach these networks based on the metric. The **med** value should be set by outbound filters of **BGP1** and **BGP2**.

For Policy 7, a common practice is for AS200 to prepend its own AS number to all updates sending to AS190. This would discourage incoming traffic from going through AS190 because everything else being equal BGP will select the route with the shortest AS Path.

Based on these requirements, administrators then choose the appropriate configuration commands and parameter values for each router to satisfy them. Figure 8 and Figure 9 give a snapshot of the working configuration of **BGP1** and **BGP2**.

The configuration errors that can arise include (in fact, we inadvertently made most of them in setting up our network):

1. Forget to configure the static route to the loopback interface of **BGP1** and **BGP2**. Neighbor address becomes unreachable. BGP session could not be established.
2. Forget to specify the **update-source** in the **neighbor** command. TCP connection is rejected and BGP session could not be established.
3. Forget the AS path access list. AS200 becomes a transit AS of AS180 and AS190.
4. Incorrect route maps and filters due to misunderstanding of the syntax of regular expression and meaning of path attributes. For example, a lower **med** value is considered better, which is in contrast to a higher **local-preference** is favored in the route selection process.
5. For Policies 5 and 6, **BGP1** and **BGP2** should give, respectively, more and less favorable values to **med**. It is entirely possible that both give equally favorable values.
6. For Policy 7, AS200 should prepend its own AS number in all updates to AS190, and this rule should be enforced both at **BGP1** and **BGP2**. If it is forgotten at one router, Policy 7 will not be implemented.

```
!
hostname BGP2
!
router bgp 200
   no synchronization
   network 200.12.2.0
   neighbor 180.200.2.2 remote-as 180
   neighbor 180.200.2.2 route-map SETMEDOUT out
   neighbor 180.200.2.2 filter-list 1 out
   neighbor 190.200.2.2 remote-as 190
   neighbor 190.200.2.2 route-map SETLOCALIN in
   neighbor 190.200.2.2 route-map SETASPATH out
   neighbor 190.200.2.2 filter-list 1 out
   neighbor 200.12.1.1 remote-as 200
   neighbor 200.12.1.1 Update-source Loopback0
!
ip as-path access-list 1 permit ^$
ip as-path access-list 2 permit 172$
!
access-list 1 permit 200.12.2.0 0.0.0.255
route-map SETLOCALIN permit 10
   match as-path 2
   set local-preference 300
!
route-map SETLOCALIN permit 20
   set local-preference 100
!
route-map SETMEDOUT permit 10
   match ip address 1
   set metric 10
!
route-map SETMEDOUT permit 20
   set metric 30
!
route-map SETASPATH permit 10
   set as-path prepend 200 200
!
ip route 200.12.1.0 255.255.255.0 200.12.3.1
!
```

Figure 9: **BGP2** configuration.

This example shows that manual compilation of high-level requirements into low-level configuration is a rather demanding and error-prone process. Even for a small network, the resulting configuration is already complex. It is not so obvious how each individual commands relate to intended policies. A large network has many more routers to manage and much more

complicated policies in place. The service requirement also changes more frequently. It will be even harder for the administrator to keep the mental map of how each policy is effected on each individual routers, and make sure adding or modifying devices and services does not violate existing requirements.

Using Service Grammar, the global requirements of AS200 can be described as shown in Table 2.

The description is concise and hides most low-level details. More importantly, it highlights the constraints spanning multiple routers that have to be enforced. Keeping track of such global constraints in low-level configuration language would be much harder. The diagnosis engine can effectively identify the configuration errors listed above. For instance, the first two errors will result in ibgp_session_test to fail. Missing AS path access list can be detected by link_to_provider. Similarly, misconfigured route-maps are caught by policy grammar rules.

Related Work

Our system provides a language to express the logical structure of an AS and its BGP policies. It and can automatically check expressions in this language against router configurations and thereby provide a useful diagnosis service, which has not been available to date. The main difference between this approach and previous diagnosis systems is that we can describe the BGP architecture of an AS in a high level language and check that it has been correctly configured. If someone changes a router configuration, he can just run the diagnosis again to ensure that the logical structure and policies have not been violated. In Netsys [8], there is no way to describe the administrator's *intention*, i.e., *network-specific* policies and structure. It just runs a collection of network-invariant tests.

Routing Policy Specification Language (RPSL) [12] allows a network operator to specify routing policies in a high-level language. Given sufficient details, low-level router configurations can be generated from the description. RPSL shares some common views with our approach. We feel Service Grammar is better suited for diagnosing configuration errors because of its expressive power. In the short run, we believe diagnosis is even more important than provisioning because it gives administrators the desirable level of control and predictability, and can be used immediately.

It is natural to extend our system for provisioning. Given a comprehensive service specification, it can be compiled into vendor-neutral component configurations. Vendor-specific adaptors can then be applied to generate low-level router configuration commands. Reference [4] demonstrates a system that implements Service Grammar rules in Prolog. Because of the relational nature of Prolog, service specification simultaneously serves provisioning purposes. Another

```
bgpAS200 :-
    AS200basicConnectivity, AS200policies.
AS200basicConnectivity :-
    ibgp_session(BGP1, BGP2, AS200),
    ebgp_session(BGP1, AS200, BGP3, AS180),
    ebgp_session(BGP2, AS200, BGP3, AS180),
    ebgp_session(BGP2, AS200, BGP4, AS190),
    as_full_mesh({}, {BGP1, BGP2}, AS200).
AS200policies :-
    policy1, policy2, policy3, policy4, policy5, policy6, policy7.
policy1 :-
    route_originate({200.12.1.0/24, 200.12.2.0/24}, AS200).
policy2 :-
  provider_AS(AS200,AS180),
    link_to_provider(BGP1, BGP3),
    link_to_provider(BGP2, BGP3),
  provider_AS(AS200,AS190),
    link_to_provider([BGP2, BGP4).
policy3 :-
    preferred_outgoing_link(BGP2, BGP4, AS172).
policy4 :-
    preferred_outgoing_link(BGP1, BGP3, AS180).
policy5 :-
    preferred_neighbor_entry(BGP1, BGP3, 200.12.1.0/24).
policy6 :-
    preferred_neighbor_entry(BGP2, BGP3, 200.12.2.0/24).
policy7 :-
    preferred_incoming_link(BGP1, BGP3, ALL), preferred_incoming_link(BGP2, BGP3, ALL).
    preferred_incoming_link(BGP2, BGP3, ALL), preferred_incoming_link(BGP2, BGP3, ALL).
```

Table 2: AS200 service grammar.

system that performs a set of specialized provisioning tasks is described in [13].

Reference [14] studies IBGP routing anomalies and proposes sufficient conditions that guarantee correctness. These conditions can be incorporated into our Service Grammar as policy templates. Reference [10] presents an empirical study of BGP misconfigurations. Several classes of errors, such as reliance on upstream filtering, forgotten filter, incorrect summary, bad route map, etc., are caused by simple high-level policies that are not obvious for operators to express at the CLI level. Our system would reduce these types of errors given the high-level, system-wide requirements.

Summary

Our Service Grammar system enables a new way of configuration error diagnosis in a distributed environment. Network administrators can express the global requirements in a high-level language. The diagnosis engine then systematically verifies if the expressions in this language are satisfied by device configurations in a top-down fashion. Our system highlights global reasoning – i.e., why a group of components fail to jointly compose the intended service – rather than why a single component fails.

We demonstrate how to use such a system to diagnose BGP configuration errors in an AS. Previous solutions to diagnosing configuration errors have been network invariant in that they contain a fixed set of constraints that must be satisfied by every BGP network. However, BGP requirements such as logical architecture and policies differ widely from one network to another. It is not possible in previous techniques to check if component configurations implement the network-specific requirements. Our language consists of a small set of abstractions that can be composed to describe most BGP features.

One limitation of this approach is that Service Grammars and diagnosis procedures must be developed for each protocol of interest. We have prototyped Service Grammars for RIP, OSPF, BGP, BGP/MPLS, PIM, GRE, IPSEC, DiffServ and the Spread group communication protocol [15]. The difference between grammars tends to be significant because they are very protocol-specific. Based on our experience, the amount of effort required to develop the Service Grammar for a protocol is not terribly large. The notion of correct configuration is already implicit in the definition of protocols since their intended use is a part of the definition. The job of the Service Grammar designer is essentially to make this "configuration logic" explicit by analyzing these definitions.

The challenge for end users is that they need to go through another learning curve, and may still write incorrect specifications in this language. However, we believe the chances of errors should be lowered in this high-level language as oppose to low-level configuration commands.

Author Information

Xiaohu Qie is a Ph.D. candidate in the Department of Computer Science at Princeton University. His research interests cover the general areas of operating systems, networking and security. He received his B.S. degree from Tsinghua University, Beijing, China, in 1998, and M.A. degree from Princeton University, Princeton, NJ in 2000. He is a student member of USENIX, IEEE, and ACM SIGOPS. He can be reached at qiexh@CS.Princeton.EDU .

Sanjai Narain is a Senior Research Scientist in the Information Assurance and Security Department in Telcordia Technologies' Applied Research Area. He developed the Service Grammar technique that makes systems integration via configuration much more efficient than possible today. This technique has been used in DARPA projects on synthesis of dynamic coalitions and secure, survivable information systems. It has also been integrated into a Telcordia operations support system for IP Virtual Private Networks. In 2003, the dynamic coalitions project won a DARPA award for technology transfer to the Army's Future Combat Systems program. Earlier, he developed the DR. DIALUP software for reducing help-desk costs of Internet Service Providers, a DSL loop qualification tool that is the basis for a successful Telcordia service, and a SONET conformance-testing tool that was used by Telcordia's professional services. Prior to joining Telcordia, he designed logic-based simulation techniques at The RAND Corporation. His background is networking, symbolic logic, automated reasoning and programming languages. He obtained a Ph.D. in Computer Science from University of California, Los Angeles, in 1988, an M.S. in Computer Science from Syracuse University, in 1981, and a B.Tech in Electrical Engineering from Indian Institute of Technology, New Delhi, in 1979. He can be reached at narain@research.telcordia.com .

References

[1] Lampson, Butler, "Computer Security in the Real World," *Proceedings of Annual Computer Security Applications Conference*, 2000.

[2] Horn, Paul, *Autonomic Computing: IBM's Perspective on the State of Information Technology*, http://www.research.ibm.com/autonomic/manifesto/autonomic_computing.pdf.

[3] Barton, M., D. Atkins, S. Narain, D. Ritcherson, K. Tepe, "Integration of IP Mobility and Security For Secure Wireless Communications," *Proceedings of IEEE International Communications Conference*, New York, NY, 2002.

[4] Narain, S., B. Coan, V. Kaul, K. Parmeswaran, W. Stephens, "Building Autonomic Systems Via Configuration," To appear in *Proceedings of IEEE Autonomic Computing Workshop*, June, 2003.

[5] Narain, S., A. Shareef, M. Rangadurai, "Diagnosing Configuration Errors in Virtual Private Networks,"

Proceedings of IEEE International Communications Conference, Helsinki, Finland, 2001.

[6] Narain, S., R. Vaidyanathan, S. Moyer, W. Stephens, K. Parmeswaran, A. Shareef, "Middleware for Building Adaptive Systems Via Configuration," *Proceedings of ACM SIGPLAN Workshop on Optimizing Middleware*, Salt Lake City, UT, June, 2001.

[7] Rekhter, Y. and T. Li, "A Border Gateway Protocol 4 (BGP-4)," *Request for Comments 1771*, March 1995.

[8] Netsys, http://www.cisco.com/warp/public/cc/pd/nemnsw/nesvmn/index.shtml.

[9] *BGP Commands*, http://www.cisco.com/univercd/cc/td/doc/product/software/ios120/12cgcr/np1_r/1rprt1/1rbgp.htm.

[10] Mahajan, Ratul, David Wetherall, and Tom Anderson, "Understanding BGP Misconfigurations," *Proceedings of ACM SIGCOMM*, August, 2002.

[11] Gao, L. and J. Rexford, "Stable Internet Routing Without Global Coordination," *Proceedings of ACM SIGMETRICS*, June, 2000.

[12] Alaettinoglu, C., C. Villamizar, E. Gerich, D. Kessens, D. Meyer, T. Bates, D. Karrenberg, and M. Terpstra, "Routing Policy Specification language (RPSL)," *Request for Comments 2622*, June, 1999.

[13] Gottlieb, Joel, Albert Greenberg, Jennifer Rexford, and Jia Wang, "Automated provisioning of BGP customers," In submission, December, 2002.

[14] Griffin, T. G. and G. Wilfong, "On the Correctness of IBGP Configuration," *Proceedings of ACM SIGCOMM*, August, 2002.

[15] *The Spread Toolkit*, http://www.spread.org/.

Splat: A Network Switch/Port Configuration Management Tool

Cary Abrahamson, Michael Blodgett, Adam Kunen, Nathan Mueller, and David Parter –
University of Wisconsin – Madison

ABSTRACT

We present the design and implementation of Splat, a tool for managing network edge switch ports and port-to-host configurations. We discuss the need for such a tool as part of a major network upgrade, and our discovery that most existing tools ignore this area or approach it from a core network point of view.

Important design considerations include the current procedures and habits of our system administration team, using open source components, and the tool's incremental development and deployment. In particular, the requirement for accurate configuration information in a database proves to be an effective means of enforcing correct procedures.

Splat enables us to administer an increasingly complex network by providing a simple interface for routine tasks as well as better diagnostics, reporting, and monitoring. Its reliable configuration management enhances the security and performance of our network.

Introduction

Modern network switches continue to become more complex to configure and manage as functionality is moved from the network core to edge devices. As a result, administrators of large Local Area Networks (LANs) face significant challenges, particularly if they utilize features such as Virtual LANs (VLANs) or per host Quality of Service. Most existing network management systems do not focus on the edge devices of LANs. Instead, they are designed for network specialists administrating the core devices of Wide Area Networks (WANs). This has left many desktop and server administrators who are responsible for moves, adds, and changes involving edge devices to manually configure and maintain them because they lack adequate tools.

There are several well-known problems with manually configuring devices. First, the amount of time the network specialist must devote to reconfiguring switch ports for each workstation or server change is significant. This can also lead to bottlenecks as other administrators wait for each network reconfiguration to be completed. A large volume of manual changes also increases the probability of error with the resultant increase in downtime and troubleshooting.

In this paper, we discuss the design and implementation of Splat – a switch port configuration management tool.[1] Unlike most existing network management tools, Splat focuses on the configuration management of edge devices, applying a traditional system administration configuration management approach to this area.

Configuration Management

In recent years, large scale infrastructure configuration management has been a topic of interest to many system administrators [3, 4, 30, 15, 6]. Organizations are now deploying comprehensive configuration management systems for many areas of their infrastructure such as workstations, servers, and account management. In the traditional network management community, the emphasis has been on provisioning, bandwidth, core devices, inter-device links, and fault management [9, 24, 29, 23, 16].

Those who address the network management of LANs have tried to take a similar approach. As Kevin Dooley states in *Designing Large-Scale LANs* [10]:

> However, remember the physical tracking side of configuration management, especially if you deal with the configurations of Layer 2 devices such as hubs and switches. If network managers have accurate information about physical locations, MAC addresses, and cabling for end devices such as user workstations, then they can easily handle hardware moves, adds, and changes. In most organizations, business requirements force network administration to respond quickly and efficiently to requests for end-user moves and service changes. However, the cabling and hardware records are usually out-of-date, so every small move requires a technician to visit the site and carefully document the equipment and cabling. This process is expensive and slow.

Having out-of-date records is a problem, but not one that is unique to network management. System administrators deal with the same issue in designing and deploying configuration management systems for workstations and servers. As Dooley points out, if the data is not current it cannot be used. If it is not being used then overworked system administrators have no incentive to update the data. This chicken-and-egg situation is the bane of every system administrator trying to introduce system configuration management to a site.

[1] During design discussions, we needed a name for the system. The placeholder name "Splat" stuck.

The best way to break this cycle is to integrate a single definitive data source into operational tools. When accurate data is necessary for the operation of a site it reverses the situation. This poses a twofold challenge for the system designer: 1) Data updates must be as easy and painless as possible for the staff. 2) The benefits of the new system must be significant.

Dooley continues:

> Unfortunately, no software can solve this problem; it is primarily a procedural issue. Technicians making changes have to keep the records up-to-date, and the cabling and patch panels have to be periodically audited to ensure accuracy of the records.

We agree that this area of network management is primarily a procedural one. However, such procedures are best implemented using software tools that validate input data and ensure integrity through traditional system administration practices.

Background

The University of Wisconsin's Computer Sciences Department network consists of approximately 1500 computers using 2000 IP addresses on 50 routed subnets. All system administration tasks (including desktop support, software installation, account management, and network management) are handled by one group. This group has nine full-time staff and 12 part-time undergraduate students. There is some specialization in both the full-time staff and the student staff. But, everyone is expected to be able to perform some common tasks. For example, all of the student staff are expected to do routine tasks such as moving workstations from one office to another, which often involves IP address changes.

We previously had a small number of routers and a large number of relatively simple (and "mostly unmanaged") 100base-T ethernet switches with no VLANs. In that situation, it was reasonable to not use any network configuration tools since the only devices that needed ongoing configuration were the routers and firewalls, each of which was a special case. Router and firewall configurations were manually preserved using RCS [28].

Motivation and Requirements Identification

We began a major network infrastructure upgrade in the summer of 2002. All routers and the "mostly unmanaged" ethernet switches were to be replaced with new routers and switches.

The new network consists of three core routers and edge switches in the data center and edge switches in Intermediate Distribution Frames (IDFs)[2] throughout the building. Multiple routed networks enable traffic separation and functional grouping of computers. Each edge switch has redundant uplinks to the core routers. IEEE 802.1Q [17] VLANs are used to implement the separate routed networks on one set of equipment.

The previous network had only three managed devices; the new network would have over 50 devices that required active management. It became clear that we needed a switch configuration management tool.

Requirements

A switch configuration management tool must satisfy a number of requirements to be useful to our staff:

1. It must enable administrators with little networking experience to perform routine tasks without having to log into edge switches and make manual configuration changes. These tasks include adding, removing, and swapping workstations in offices throughout the department and making hardware changes such as replacing ethernet cards.
2. The tool should provide live port status and statistics from switches without requiring administrators to log into devices manually.
3. It should report reliable, up-to-date information on which host is connected to which switch port for troubleshooting and general auditing. Previously this would have been done by sending a staff person to an IDF and computer location to manually take an inventory of the switch ports, patch panels, and data jacks.
4. The tool should track network device configuration changes using a version control system such as CVS [7].
5. It must allow for different policies and configurations on different VLANs.
6. It must be possible to configure any VLAN in our network on any edge port of any switch.
7. The tool should be easy to integrate with our existing site configuration management infrastructure, and it must be scalable and extendable for additional features as our network continues to evolve.
8. Use of the tool should be easily integrated with existing staff procedures and habits, causing as little disruption as possible.

Additional Features

Although not strictly required, some additional features would enhance the functionality of the tool:

1. Allowing only a specified MAC address on a given switch port would enhance the security of the network.
2. The tool should automatically generate configuration files for other network tools.

Alternatives Considered

System administrators have a propensity for reinventing the wheel. Before creating a new system, we

[2] Network jargon for a wiring closet. More specifically, in a structured wiring system, the rack where end devices are cross-connected into the rest of the network.

looked for existing solutions. A number of proprietary and open source options were investigated.

As previously noted, most network management systems do not support the type of edge-device port configuration management we were looking for. Systems such as HP OpenView, IBM Tivoli NetView, and GxSNMP are best suited for management of WANs or campus core networks, not edge device ports. OpenNMS is primarily a fault and performance management system. We found that the LANdb project ("The Network Management Database") [18] addresses similar areas as Splat, but it appears that development has been idle since July 2000.

We did not find any systems that met our requirements, so we decided to develop Splat.

Tool Design and Implementation

Splat is a command line tool, written in Perl [25], and a related set of Perl modules. It interfaces with a PostgreSQL [26] network configuration database using the Perl Database Interface module (DBI) [8]. It was implemented without any special PostgreSQL features and it would be easy to use a different relational database system in its place.

Splat interacts with switches using the RANCID [27] configuration tool. It has been tested and used on Linux and Solaris systems. Splat has not been tested on Windows.

Splat generates the appropriate switch configuration commands from templates and uses RANCID to apply those commands to the switches. RANCID then retrieves the complete configuration from the switch and stores it in a CVS repository (see Figure 1). This enables administrators to view and retrieve previous configuration versions.

Network Configuration Database

Managing site configuration information with a relational database has been a common strategy of system administrators for some time [12, 13, 14]. Splat applies this strategy in managing the edge devices of a network. Its network configuration database contains both static information (such as data jack locations) and dynamic information (such as current hostname/data-jack assignments). The details of the database design are covered in the next section.

Ideally, the network configuration database is part of a larger configuration management system, including a parts inventory of workstations components (for example, ethernet cards) and other information such as operating system and security policies. We import inventory data from our existing inventory database.

Sites without an inventory database need to enter ethernet card and hostname information into the Splat database. Splat includes scripts to initialize other information in the database (such as IDFs and jack-numbers) according to the naming conventions of the site.

Tables and Data Relationships

The major data tables and relationships are shown in Figure 2. The primary abstraction in Splat is the relationship between a particular host network interface and a particular data jack – "where the host meets the network." It is entered in the *hosts* table in the Splat database. The computer's network adapter is represented as an inventory part number and interface number (to support multi-port network cards). The data jack is represented as an (IDF, jacknumber) pair. At our site, data jacks are numbered per IDF. Other naming schemes will work as long as the (IDF, jack-number) pair is unique.

In the IDF, each switch port is patched to a data jack. This is represented in the *port_mapping* table. The *jacks* table reflects the building installed wire between the IDF patch panel and office data jack. If enough switch ports are available, the use of VLANs and trunking reduces the need for re-patching between switch ports and the IDF patch panel. To generate a *generic* port configuration script, Splat consults the *mac_address* and *vlans* database tables. Splat also needs the IP address, which could be stored in the configuration database or retrieved by a Domain Name System query on the hostname.

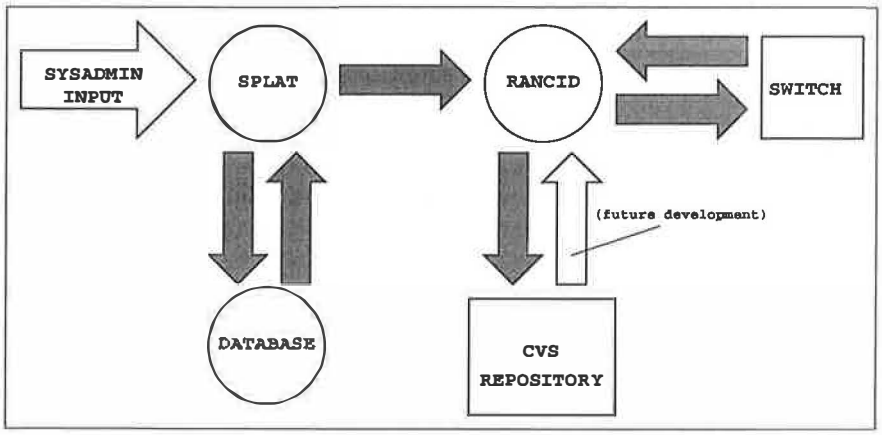


Figure 1: Splat processes and related components.

Additional database tables hold IDF data (names and locations), switch data (names, locations, and interface inventory), and "glue" to interface to our existing inventory database.

MAC Address Locking

On most of our production networks, the generated configuration includes commands to "lock" the switch port to the MAC address of the attached workstation. This serves as an integrity check on the configuration and also makes it more difficult for the casual "bandwidth borrower" to unplug a supported workstation and use the data jack for their laptop.

Not all switches feature the ability to lock a port to a specific MAC address. For those that do, the commands to configure MAC-locking vary from vendor to vendor. Splat can handle this by using different configuration templates for each vendor to generate command scripts to run on the appropriate switch.

Once the port is configured for only one MAC address, the switch handles violations depending on the switch and the specific configuration options used. In our case, the Cisco switches are configured to *restrict*, which means that packets from other MAC addresses are dropped, and SNMP traps and syslog messages are generated about the event (monitoring SNMP traps and syslog messages is outside the scope of Splat).

Generating a MAC-lock configuration uses the *mac_address* table, as shown in the middle and bottom of Figure 2.

Port Policies

Uplink ports use a different configuration template than edge-device ports. The port configuration does not use MAC address locking or assign a VLAN, but instead sets the port to *trunk* mode and DOT1Q encapsulation. Other port policies are *broken, reserved,* and *generic* (available for use).

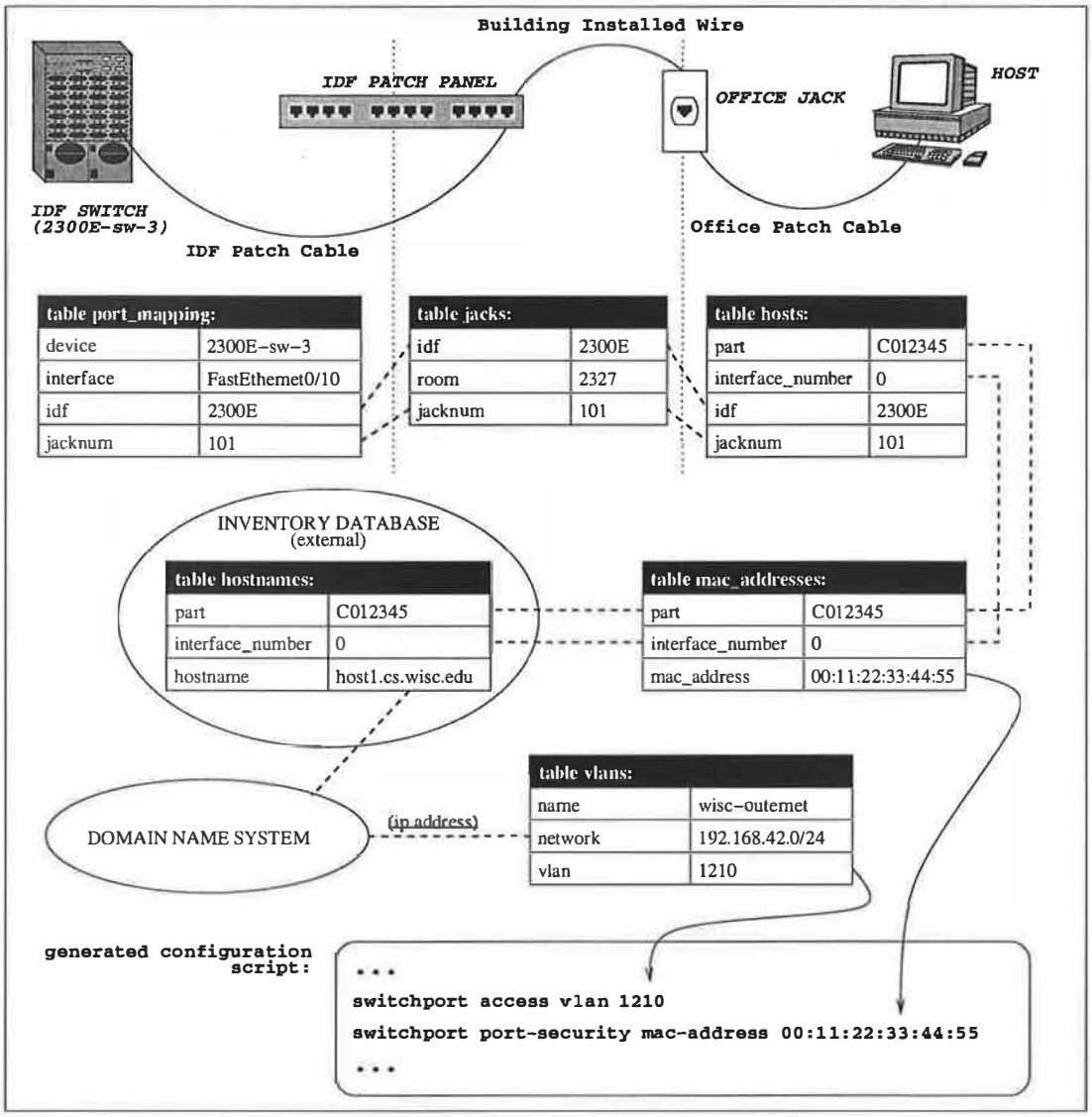


Figure 2: Data relationships.

Per-switch-interface (port) policies are in the *switch_interfaces* table (not shown in Figure 2).

VLAN Table

We use a *vlans* table instead of an IP address to VLAN scheme (such as using a "subnet" number) to eliminate any assumptions about IP network address allocation and avoid conflicts with reserved VLAN numbers. This is shown at the bottom of Figure 2.

VLAN Policies

Not all of our VLANs use the same configuration policy. For example, the "laptop" network is designated for laptop use. It is available in certain locations for transient laptops. The VLAN policy for the laptop network does not specify "MAC-locking".[3] Also, on general-purpose networks, data jacks that are not currently *attached* to a computer are disabled in the configuration. Laptop network public data jacks are enabled, although they are not *attached* to a specific computer.

VLAN policy assignments are in the *vlan_policy* table (not shown in Figure 2).

RANCID

Splat utilizes RANCID to securely communicate with network devices, execute device commands, and maintain device configurations with CVS. Although RANCID is the "Really Awesome New Cisco conflg Differ," it actually supports a number of well known network devices in addition to Cisco switches running IOS [5]. As of RANCID version 2.2.2, this includes Bay routers, Juniper routers, Cisco Catalyst switches, Foundry switches, Redback NASs, ADC EZT3 muxes, MRTd, Alteon switches, and HP ProCurve switches. This flexibility is important as we do not know what devices we will need to support in the future.

To run commands on a device, Splat first generates a device configuration script from a template. The commands are then run on the switch via RANCID's *clogin* utility. Clogin is an Expect [19] script that automates the process of logging into devices using the facilities available on the device such as SSH [31].

For each device configuration change, RANCID makes a revision entry in CVS. It is important to note that we do not use CVS for classic revision control. While it is possible to load a previous configuration revision directly to a switch, the Splat database would not be synchronized. Instead, we use CVS revisions for logging changes and as a backup for disaster recovery if the switch needs to be reinstalled or completely reconfigured.

Using Splat

The Splat command syntax is described in Appendix A. Output from these commands are shown in Appendix B. The two most common commands are *attach* and *detach* to update which computer is connected to a particular data jack.

Desktop and Server Administrators

Splat enables desktop and server administrators to make network configuration changes that are needed when doing routine tasks such as moving or reconfiguring a computer without intervention from network specialists. When attaching a computer to a data jack, the system administrator does not need to know the MAC addresses or VLAN number, just the hostname and data jack number.

When replacing an ethernet card in a computer, the system administrator only needs to update the inventory database. Splat will see the new MAC address the next time the computer is *attached* to a data jack. The same applies when a computer changes hostname, is moved from one room to another, or is replaced by a new computer.

Information commands available to the system administrator can aid in preliminary diagnosis of host specific network problems.

Network Specialists

Compared to desktop and server administrators, network specialists often have a different view of the network and have different needs from a network management system.

Long-term and short-term planning tasks are served by the variety of information Splat makes available. For example, it is easy for network specialists to check the inventory of available switch ports and data jacks. When a research group decides to add additional computers to their desks, it is easy to check the availability of data jacks in their offices. When planning the budget, it is easy to get a site-wide inventory of available switch ports.

When troubleshooting network problems (short of total connectivity failure) it is helpful to have information about a computer's switch port connection, the switch port status and statistics, and the switch global status and statistics. The same holds true for all of the switches in the data path between the problem computer and the "other" end. Splat can present all of this information based on only one unique element of the configuration (typically either a hostname, data jack, or switch port). So, the network specialist does not have to untangle all of the different aspects of the configuration for that element.

We make extensive use of Cricket [1] for gathering and presenting network performance and utilization data, and NetSaint [22] for status/fault monitoring. Using the Splat database, we have automated the generation of configuration files for both[4]

[3] Laptop network authentication and authorization are not part of Splat. We use authipf [2].

[4] Note: Nagios [21] is the successor to NetSaint, and we will generate Nagios configuration files when we switch from NetSaint.

Deployment and Experiences

Splat was developed on a test network composed of a subset of the switch setup that was to be used department-wide. It was deployed one network at a time as existing switches were replaced. As each of the first few networks were converted, we assessed the process and made appropriate adjustments to the code.

There was some hesitation by the staff when they started using Splat. During the transitional phase, it was unclear which networks were under Splat configuration, and which were to be done "the old way." This was a communication problem, not a Splat problem.

Networks were converted to Splat by a handful of student staff (including, of course, the students who specialize in the network and had been involved in planning and designing Splat). They soon became very familiar with Splat, and were willing to work around any bugs they found (which were usually fixed within hours). The vast majority of the time the only command being used was *attach*, as workstations moved from the old (unmanaged) network to the new network with Splat.

Using Splat requires that the inventory database be correct and current. As can be expected, a number of data entry errors were found when Splat either refused to make a connection (no network part in the inventory) or configured the wrong MAC address (wrong part in the inventory). Once the staff became aware of the problem, these data errors were fairly easy to fix on a case-by-case basis.

The requirement that the inventory database be current also caused some issues, as staff who were not in the habit of updating the inventory in a timely manner were forced to make the update before being able to attach a computer to the network.

We experienced a few interesting events throughout the deployment. For example, during a rack cleanup in the data center, it was necessary to move the database server that included the Splat database from one network switch to another. It was immediately apparent that *detaching* the Splat database server was a mistake, as it was no longer possible to run the Splat commands necessary to re-*attach* it to the network.

Also, we had been aware for some time that a few rogue users had been occasionally disconnecting workstations from the network in order to use the data jacks for other computers (most likely laptops). We did not have adequate tools to catch them and it was not considered a major problem. But, it served as a catalyst for the MAC-locking feature. As soon as we deployed Splat with MAC-locking, a few MAC-lock violation alerts were generated, but no one ever complained or even asked about the change in functionality.

Additional Splat Tools and Future Development

We are continuing the development of Splat and related tools. An existing related tool is splat2cricket which generates configuration files for monitoring switches with Cricket. We are in the process of creating DNS [20] and DHCP [11] scripts that will automatically generate DNS and DHCP configuration files from the Splat network configuration database.

Needs we expect to address in the near future include multi-level access control, configuration of our core routers (in addition to edge switches), and better support for reinstalling a complete switch configuration.

Multi-level Access Control

Multi-level access control will be necessary in our department. At least one of our research projects needs the flexibility to frequently reconfigure several test networks without intervention from the system administration staff. Sites with a different system/network administration organizational model, or sharing administration of a network between different organizations, would also need it.

One approach is to control access based on the set (network device, VLAN, host). Splat, as currently implemented, directly accesses the database and configuration files. Database and file system access controls limit access to the system administration staff. Multi-level access control will require finer-grained controls than are provided by either, so it will have to be implemented in Splat.

Most likely, this will require a client-server implementation, with the server enforcing fine-grained access control. This may be the basis for web or other graphical user interfaces.

Core Routers

Our core routers are currently configured manually using RCS [11] to preserve configuration revisions. We would like to adapt Splat to manage these devices as well. This would require adding a new *router* port type to the *switch_interfaces* table and the code and templates to support it.

Reconfiguration

Currently, we can manually retrieve a switch configuration from the CVS repository when it is necessary to completely reconfigure a switch (for example, when a switch is replaced). We would like to add automated configuration recovery and initial switch configuration for new switches.

Availability

Splat is available for download from http://www.cs.wisc.edu/csl/projects/splat.

Conclusion

We presented a method for managing network edge switch ports and port-to-host configurations in the form of a relatively simple tool. Most network configuration management tools do not cover this area. Instead of relying on manual procedures, a better

way to approach this problem is by applying traditional configuration management techniques. In particular, the requirement for accurate configuration information in a database proves to be an effective means of enforcing correct procedures.

Considering the existing procedures of a site in the tool's design, and providing a simple interface, makes it possible for administrators of varying network experience to perform routine tasks. This tool enhances our ability to monitor and troubleshoot problems as our network continues to evolve.

Acknowledgements

This work would not have been possible without the help and support of the Computer Systems Lab staff.

Author Information

All of the authors work for the Computer Systems Lab at the University of Wisconsin Computer Sciences Department.

Cary Abrahamson is a System Administrator at the Computer Systems Lab. His professional interests include network management and security. You can reach him at cary@cs.wisc.edu.

David Parter is a Senior Systems Administrator and Associate Director of the Computer Systems Lab. He also serves on the SAGE Executive Committee, and was program chair for LISA '99. His professional interests include network systems, security, configuration management, and System Administration education. You can reach David at dparter@cs.wisc.edu.

Michael Blodgett is an undergraduate student and System Administrator with the student staff. Michael can be reached at mblodget@cs.wisc.edu.

Adam Kunen is an undergraduate student and System Administrator with the student staff. Adam can be reached at ajkunen@cs.wisc.edu.

Nathan Mueller is a System Administrator with the student staff who recently graduated with a degree in Computer Science from the University of Wisconsin. You can reach him at nate@cs.wisc.edu.

References

[1] Allen, Jeff R., "Driving by the Rear-View Mirror: Managing a Network with Cricket," *First Conference on Network Administration (NETA '99)*, pp. 1-10, Santa Clara, CA, USENIX, April 7-10, 1999.

[2] Beck, Robert, "Dealing with Public Ethernet Jacks – Switches, Gateways, and Authentication," *13th Systems Administration Conference (LISA '99)*, pp. 149-154, USENIX, December, 1999.

[3] Burgess, Mark, "A Site Configuration Engine," *Computing Systems*, Volume 8, pp. 309-337, USENIX, Summer, 1995.

[4] Burgess, Mark, "Computer Immunology," *Twelfth Systems Administration Conference (LISA '98)*, p. 283, Boston, Massachusetts, USENIX, December 6-11, 1998.

[5] Cisco Systems, Inc., *Cisco IOS*, http://www.cisco.com/univercd/cc/td/doc/product/software/ios123/index.htm.

[6] Cons, Lionel and Piotr Poznanski, "Pan: A High-Level Configuration Language," *Sixteenth Systems Administration Conference (LISA '02)*, pp. 83-98, USENIX,, November 2002.

[7] *The Concurrent Versions System*, http://ccvs.cvshome.org.

[8] Descartes, Alligator and Tim Bunce, *Programming the Perl DBI*, O'Reilly, February, 2000.

[9] Dooley, Kevin, *Designing Large-Scale LANs*, O'Reilly, January, 2002.

[10] Dooley, Kevin, *Designing Large-Scale LANs*, Chapter 9.1.1, p. 274, O'Reilly, January, 2002.

[11] Droms, Ralph, *Dynamic Host Configuration Protocol,* RFC 2131, March, 1997.

[12] Finke, Jon, "Automating Printing Configuration," *LISA VIII Conference Proceedings*, pp. 175-183, San Diego, CA, USENIX, September 19-23, 1994.

[13] Finke, Jon, "Institute White Pages as a System Administration Problem," *10th Systems Administration Conference (LISA '96)*, pp. 233-240, Chicago, IL, Usenix, September 29-October 4, 1996.

[14] Finke, Jon, "Automation of Site Configuration Management," *Eleventh Systems Administration Conference (LISA '97)*, p. 155, San Diego, CA, USENIX, October 26-31, 1997.

[15] Finke, Jon, "Monitoring Application Use with License Server Logs," *Eleventh Systems Administration Conference (LISA '97)*, p. 17, San Diego, CA, USENIX, October 26-31, 1997.

[16] *GxSNMP*, http://www.gxsnmp.org/.

[17] "IEEE Standards for Local and Metropolitan Area Networks: Virtual Bridged Local Area Networks," *IEEE Standard 802.1Q-1988*, 1998.

[18] *LANdb: The Network Management Database*, http://landb.sourceforge.net/about.shtml.

[19] Libes, Don, "expect: Curing Those Uncontrollable Fits of Interaction," *USENIX Summer 1990 Conference Proceedings*, pp. 183-192, USENIX, 1990.

[20] Mockapetris, P., *Domain Names – Concepts and Facilities*, STD 13, RFC 1034, November, 1987.

[21] *Nagios*, http://www.nagios.org/.

[22] NetSaint, http://www.netsaint.org.

[23] openNMS, http://www.opennms.org.

[24] *hp OpenView*, http://www.managementsoftware.hp.com/.

[25] *Perl*, http://www.perl.com.

[26] *PostgreSQL*, http://www.postgresql.org.

[27] *RANCID – Really Awesome New Cisco conflg Differ*, http://www.shrubbery.net/rancid/.

[28] Tichy, Walter F., "RCS – A System for Version Control," *Software – Practice and Experience*, Vol. 15, Num. 7, pp. 637-654, 1985.

[29] IBM Tivoli NetView, http://www.tivoli.com/products/index/netview/.

[30] Traugott, Steve and Joel Huddleston, "Bootstrapping an Infrastructure," In *Twelfth Systems Administration Conference (LISA '98)*, p. 181, Boston, MA, USENIX, December 6-11, 1998.

[31] Ylonen, Tatu, "SSH – Secure Login Connections Over the Internet," *6th USENIX Security Symposium*, pages 37-42, USENIX, San Jose, CA, July 22-25 1996.

Appendix A: Splat Syntax

As stated earlier, Splat is a command line tool. In order to meet the requirement of enabling administrators with little networking experience to make routine changes to the LAN, a simple syntax was employed. The basic operations are:

- Attaching a host to a jack.
- Detaching a host from a jack.
- Swapping two hosts.
- Getting information on a hostname, jacknumber, IDF, switch, or room.

Syntax Summary:

```
splat [ -a|--attach [ -b|--brief ] [ -n|--nolock ] hostname jacknumber [ vlan ] ]
      [ -c|--commands switch ]
      [ -d|--detach [ -b|--brief ] hostname ]
      [ -h|--help ]
      [ -s|--swap [ -b|--brief ] hostname1 hostname2 ]
      [ -i|--info [ -b|--brief ]  [ hostname | idf | jacknumber |  switch ]]
```

Appendix B: Examples

Host Information

When given the name of an attached host, splat -i prints the data jack connection information and available switch port statistics for the host. Switch port statistics vary depending on the switch type. Below is an example involving a Cisco 3500 switch.

```
$ splat -i host1.cs.wisc.edu
-------------------------------------------------------------------------
SPLAT NETWORK DATABASE                          Mon Jun 30 10:39:55 2003
-------------------------------------------------------------------------
 Hostname: host1.cs.wisc.edu
     Part: C013317                                  Bldg: CS
Interface: 0                                        Room: 2327
      MAC: 00:e0:18:71:91:0b                        Note: NA
IDF    Jack  Switch                                 Portnumber
-------------------------------------------------------------------------
2300E  218   2300E-sw-3.cs.wisc.edu                 FastEthernet0/42
-------------------------------------------------------------------------

Available Switch Statistics
-------------------------------------------------------------------------
FastEthernet0/42 is up, line protocol is up (connected)
  Hardware is Fast Ethernet, address is 000a.8aac.73aa (bia 000a.8aac.73aa)
  Description: host1.cs.wisc.edu via 2300E-218
  MTU 1500 bytes, BW 100000 Kbit, DLY 100 usec,
     reliability 255/255, txload 1/255, rxload 1/255
  Encapsulation ARPA, loopback not set
  Keepalive set (10 sec)
  Full-duplex, 100Mb/s
  input flow-control is off, output flow-control is off
  ARP type: ARPA, ARP Timeout 04:00:00
  Last input never, output 00:00:01, output hang never
  Last clearing of "show interface" counters never
  Input queue: 0/75/0/0 (size/max/drops/flushes); Total output drops: 0
  Queueing strategy: fifo
  Output queue :0/40 (size/max)
  5 minute input rate 0 bits/sec, 0 packets/sec
  5 minute ouxtput rate 0 bits/sec, 0 packets/sec
     1167225 packets input, 545465471 bytes, 0 no buffer
     Received 66 broadcasts, 0 runts, 0 giants, 0 throttles
     0 input errors, 0 CRC, 0 frame, 0 overrun, 0 ignored
     0 watchdog, 0 multicast, 0 pause input
     0 input packets with dribble condition detected
     1338665 packets output, 968728968 bytes, 0 underruns
     0 output errors, 0 collisions, 1 interface resets
     0 babbles, 0 late collision, 0 deferred
     0 lost carrier, 0 no carrier, 0 PAUSE output
     0 output buffer failures, 0 output buffers swapped out
```

Switch Information

2300E-sw-3.cs.wisc.edu is the hostname of an IDF edge switch. When given a switch name, the splat -i command shows information about the switch:

```
$ splat -i 2300E-sw-3.cs.wisc.edu
-------------------------------------------------------------------
SPLAT NETWORK DATABASE                          Mon Jun 30 10:42:09 2003
-------------------------------------------------------------------

Switch: 2300E-sw-3.cs.wisc.edu

Interface          Jacknum    Hostname
-------------------------------------------------------------------
FastEthernet0/1   2300E-201   chopin.cs.wisc.edu
FastEthernet0/2   2300E-210   tonic.cs.wisc.edu
FastEthernet0/3   2300E-211   lime.cs.wisc.edu
FastEthernet0/4   2300E-212   NA
FastEthernet0/5   2300E-213   vodka.cs.wisc.edu
FastEthernet0/6   2300E-214   ojuice.cs.wisc.edu
FastEthernet0/7   2300E-215   NA
. . .
```

Attach

The -a option is used to *attach* a computer to a data jack. The -b option requests brief output. Otherwise, each command and the resultant switch output (if any) is printed.

If the hostname is not a fully qualified domain name, an attempt is made to find one in the current domain using gethostbyname().

In this example, the host *host1.cs.wisc.edu* is moved to data jack *2300E-218*:

```
$ splat -a -b host1 2300E-218
Use fully qualified domain name host1.cs.wisc.edu? [n]/y : y

Updating the network database...
WARNING - Hostname host1.cs.wisc.edu is currently attached to jack 2300E-319.
OK to detach this connection? [n]/y : y
Host host1.cs.wisc.edu detached from jacknumber 2300E-319 in the splat database.

Updating the switch...
Host host1.cs.wisc.edu attached to jacknumber 2300E-218.

Running RANCID to collect switch config...
CVS commit for 2300E-sw-3.cs.wisc.edu config successful.
```

THE USENIX ASSOCIATION

Since 1975, the USENIX Association has brought together the community of developers, programmers, system administrators, and architects working on the cutting edge of the computing world. USENIX conferences have become the essential meeting grounds for the presentation and discussion of the most advanced information on new developments in all aspects of advanced computing systems. USENIX and its members are dedicated to:

- problem-solving with a practical bias
- fostering innovation and research that works
- communicating rapidly the results of both research and innovation
- providing a neutral forum for the exercise of critical thought and the airing of technical issues

USENIX Member Benefits

- Free subscription to *;login:*, the Association's magazine, published six times a year. *;login:* is sent in print, and is available on the USENIX Web site to all current USENIX members
- Access to papers from the USENIX conferences and symposia, starting with 1993, on the USENIX Web site
- Discounts on registration fees for the annual, multi-topic technical conference, LISA, the Systems Administration Conference, and the various single-topic symposia and workshops
- Discounts on conference proceedings and CD-ROMs from USENIX
- The right to vote on matters affecting the Association, its bylaws, and election of its directors and officers
- Savings on books, software, and periodicals: see *http://www.usenix.org/membership/specialdisc.html*

SAGE

SAGE is a Special Technical Group within USENIX. SAGE is an international membership organization whose purpose is to further advance system administration as a profession. For more information, please visit our Web site: *http://www.sage.org/*.

SAGE Member Benefits

- Discount on the registration fee for the annual LISA conference
- Access to SAGEweb, which offers many Web-based services
- The ability to join SAGE-only electronic mailing lists
- Annual System Administration Salary Survey—an annual survey of system and network administrator salaries and responsibilities—and the results of the surveys
- The SAGE Short Topics series, practical booklets covering system administration issues
- The right to vote for the SAGE Executive Committee and on other SAGE matters
- A Code of Ethics for System Administrators
- Discounts on *Sys Admin* and other publications

(Please note that *;login:* is available only as a benefit of USENIX membership.)

There are several classes of membership. You are welcome to become a member of either USENIX or SAGE, or both. Please visit our Classes of Membership Web page for further information: *http://www.usenix.org/membership/classes.html*.

USENIX & SAGE Thank Their Supporting Members

USENIX Supporting Members

❖ Ajava Systems, Inc. ❖ Aptitune Corporation ❖ Atos Origin B.V. ❖ Computer Measurement Group ❖ ❖ Interhack Corporation ❖ MacConnection ❖ The Measurement Factory ❖ Microsoft Research ❖ ❖ Motorola Australia Software Centre ❖ Sun Microsystems, Inc. ❖ Taos: The Sys Admin Company ❖ ❖ UUNET Technologies, Inc. ❖ Veritas Software ❖

SAGE Supporting Members

❖ Microsoft Research ❖ Ripe NCC ❖

For more information about membership, conferences, or publications, please visit: *http://www.usenix.org/*.

USENIX Association, 2560 Ninth Street, Suite 215, Berkeley, CA 94710 USA
Phone: 510-528-8649 Fax: 510-548-5738 Email: *office@usenix.org*

ISBN 1-931971-15-3